НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»

УДК 004.051

«До захисту допущено»
Завідувач кафедри
_____ *Стіренко С.Г.*
(підпис) (ініціали, прізвище)
"06" травня 2019 р.

Магістерська дисертація

зі спеціальності: *123. Комп'ютерна інженерія*
(код та назва напряму підготовки або спеціальності)

Спеціалізація: *123. Комп'ютерні системи та мережі*

на тему: Оптимізація конфігурацій розподілених інформаційних систем

Виконав (-ла): студент (-ка) 6-го курсу, групи ІО-71мн
(шифр групи)

Власов Максим Дмитрович _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник: доц.,к.т.н., доц., Болдак А.О.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант по нормоконтролю Куласов Ю.О.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент доц. к.т.н. каф. АСОІУ, Шишацький
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-науковою програмою
Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Спеціалізація *123. Комп'ютерні системи та мережі*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Стіренко С.Г.
(підпис) (ініціали, прізвище)
« » 20***19*** р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Власову Максиму Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Оптимізація конфігурацій розподілених інформаційних систем

Науковий керівник дисертації доц.,к.т.н., доц., Болдак А.О.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «***07***» 11 ***2018*** р. № ***4112-с***

2. Строк подання студентом дисертації 21.05.2019

3. Об'єкт дослідження конфігурація розподілених інформаційних систем

4. Предмет дослідження підходи та методи для знаходження оптимальних комбінацій складових системи за заданими параметрами

5. Перелік завдань, які потрібно розробити: 1) дослідження структури та складових розподіленої інформаційної системи; 2) аналіз наявних рішень з оптимізації конфігурацій розподілених інформаційних систем; 3) аналіз існуючих алгоритмів оптимізації і мінімізації покриття; 4) вибір підходящих підходів, проектування і розробка системи

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Професор Кулаков Ю.О.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1.	*Затвердження теми роботи*	07.11.2018	
2.	*Дослідження та аналіз теоретичної інформації*	21.11.2018	
3.	*Аналіз алгоритмів оптимізації і мінімізації покриття*	28.01.2019	
4.	*Вибір підходящих підходів і проектування системи*	25.02.2019	
5.	*Програмна реалізація системи*	11.03.2019	
6.	*Тестування розробленого способу*	02.04.2019	
6.	*Оформлення пояснювальної записки*	20.04.2019	
7.	*Передзахист*	06.05.2019	
8.	*Захист*	21.05.2019	

Студент _____ Власов М.Д.
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Болдак А.О.
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Дана магістерська дисертація присвячена проблемі оптимізації конфігурацій розподілених інформаційних систем. Складається з завдання (2 сторінки), пояснювальної записки (66 сторінок), і додатку з прототипом системи (13 сторінок). Кількість ілюстрацій – 19, кількість таблиць – 2, кількість джерел посилань – 49.

Предметом дослідження постають підходи та методи для знаходження комбінацій складових системи що найліпше відповідають заданим критеріям.

Актуальність теми.

У сучасних умовах наявна ціла серія сервісів для підняття найпростіших рішень, однак навіть в рамках одного провайдера можуть бути різні способи запуску одного і того ж з абсолютно різними підходами. Принаймні, ці підходи описані. Як же будувати складні системи, з яких цеглинок відштовхуватись - наразі це питання є невизначеним і кожен підходить до його вирішення відштовхуючись від наявного досвіду і знань, а не від загальноприйнятих стандартів чи об’єктивно найоптимальніших рішень для цього продукту в даний проміжок часу.

Метою дисертації є надання ефективного способу пошуку оптимальної конфігурації кінцевої системи вже на самих початках розробки.

Для досягнення цієї мети було проаналізувано основні складові розподіленої інформаційної системи, досліджено стан наявних рішень з оптимізації її конфігурацій, виявлено їх сильні та слабкі сторони, сформовано вимоги до нової системи, проаналізовано існуючі алгоритми оптимізації і

мінімізації покриття, вибрано підходящі і спроектовано систему що відповідає цим вимогам.

Запропонований спосіб реалізації оптимізації PIC наразі не має жодних аналогів і дозволяє кардинально збільшити швидкість розробки прототипів систем з використанням усіх новітніх можливостей і оптимізацій, що в свою чергу дозволяє використовувати найліпші практики людям, що не мають відповідних знань, а також коригувати рішення професіоналів, вказуючи на недоліки у їх системі.

Проміжні результати даної магістерської дисертації було опубліковано у «SAIT 2018» і «ICSFTI 2019».

Ключові слова: розподілена інформаційна система, оптимізація, PIC, rational unified process, RUP.

ABSTRACT

This master's dissertation is devoted to the problem of optimization of configurations of distributed information systems. The subject of the study is the approaches and methods for finding combinations of system components that best fit the specified criteria.

Actuality of theme.

In modern conditions, there is a whole series of services available to raise the simplest solutions, but even within a single provider there may be different ways of launching the same with completely different approaches. At the very least, these approaches are described. How to build sophisticated systems, from which the bricks to crawl - this question is uncertain at the moment and everyone approaches its solution based on existing experience and knowledge, and not from generally accepted standards or objectively the most optimal solutions for this product at this time.

The purpose of the dissertation is to provide an effective way to find the optimal configuration of the final system at the very beginnings of development.

To achieve this goal, the main components of the distributed information system were analyzed, the status of existing solutions was optimized for optimization of its configurations, their strengths and weaknesses were identified, requirements for the new system were formed, existing algorithms for optimization and minimization of coverage were analyzed, suitable and designed system corresponding to it requirements.

The proposed method of implementation of optimization DIS currently has no analogues and can dramatically increase the speed of development of prototype

systems with the use of all the latest features and optimizations, which in turn allows the best practices to people who do not have the relevant knowledge, as well as adjust the decisions of professionals, pointing to shortcomings in their system.

The intermediate results of this master's thesis were published in ‘SAIT 2018’ and ‘ICSFTI 2019’.

Keywords: distributed information system, optimization, DIS, rational unified process, RUP.

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

РІС	Розподілена інформаційна система
CI	Continuous Integration
CD	Continuous Delivery/Deployment
RUP	Rational Unified Process
К1	Користувач який вносить нові формальні описи елементів у систему
К2	Користувач що неформальною мовою описує вимоги до системи
DC	покриття Don't-Care [38]

ВСТУП

Зі зростанням ролі онлайн сервісів у житті людини зростають і вимоги до сервісів, які включають у себе як нові функціональні можливості, так і вимоги до швидкодії та доступності. Ці вимоги стають частиною бізнес вимог і знаходять відображення у цілях найманих працівників. Так для розробників що створюють новий функціонал та покращують старий стала важлива наявність процесів неперервної доставки та перевірки змін на наявність недоліків, більш відомі як Continuous Integration і Continuous Delivery/Deployment. Для тестувальників - написання автотестів і вбудування їх у процес автоматичної перевірки змін розробників. А перед інфраструктурними інженерами постає необхідність забезпечення роботи як CI/CD системи, так і різні оптимізації на рівні розміщення застосунків у найближчому до клієнтів датацентрі, забезпечені відмовостійкості навіть в разі форс-мажорних обставин і просто швидкість доставки нових інфраструктурних конфігурацій для нових версій застосунків що їх розробляють девелопери.

Частина цих питань є суцільно ментальною і вирішується за допомогою налагодження відповідних процесів і культури в компанії, кращі з цих практик увійшли в DevOps методологію. Однак за бажанням бути кращими, швидшими, успішнішими у бізнесу є більш конкретна мета: заробляти більше грошей. А усі ці перетворення вимагають збільшення видатків на початковій стадії і не завжди у бізнесу є відчутна грошова подушка щоб дозволити менеджерам і інженерам налаштовувати більш ефективні процеси що впродовж декількох років виливатимуться бізнесу в значні додаткові витрати. Тому **об'єктом дослідження** в межах даної магістерської дисертації є процес пришвидшення визначення необхідної кінцевої конфігурації, за допомогою чого можна пришвидшити процес прийняття правильних рішень.

Предметом дослідження постають підходи та методи для знаходження комбінацій складових системи що найліпше відповідають заданим критеріям.

Актуальність теми.

У сучасних умовах наявна ціла серія сервісів для підняття найпростіших рішень, однак навіть в рамках одного провайдера можуть бути різні способи запуску одного і того ж з абсолютно різними підходами. Принаймні, ці підходи описані. Як же будувати складні системи, з яких цеглинок відштовхуватись - наразі це питання є невизначеним і кожен підходить до його вирішення відштовхуючись від наявного досвіду і знань, а не від загальноприйнятих стандартів чи об'єктивно найоптимальніших рішень для цього продукту в даний проміжок часу.

Метою дисертації є надання ефективного способу пошуку оптимальної конфігурації кінцевої системи вже на самих початках розробки.

Для досягнення цієї мети необхідно проаналізувати основні складові Розподіленої інформаційної системи, дослідити стан наявних рішень з оптимізації її конфігурацій, виявити їх сильні та слабкі сторони, сформувати вимоги до нової системи, проаналізувати існуючі алгоритми оптимізації і мінімізації покриття, вибрати підходящі і спроектувати систему що відповідає цим вимогам.

РОЗДІЛ 1
РОЗПОДІЛЕНІ ІНФОРМАЦІЙНІ СИСТЕМИ І ВИМОГИ БІЗНЕСУ

З давніх давен перед людьми стоїть завдання отримання з меншої кількості ресурсів найбільшої користі. З розвитком технологій суспільства з'являлись все нові і нові типи ресурсів і виникла проблема вирахування користі в кожен момент часу. З цією задачею успішно впорались фінансові системи, звівши цінність кожної значної одиниці ресурсу до його еквіваленту у грошовій одиниці. Ці системи становились досить довгий період часу і лише на певному етапі розвитку суспільства. Наприклад, для повноцінного функціонування фінансової системи вимагається поява хоча б неповного письма, найбільш відоме з яких наразі зветься математичним і на самому початку складалось з десяткової системи і базових арифметичних операцій. Це обумовлено обмеженістю пам'яті і обчислювальних можливостей мозку людини [1].

Однак якщо задача співвідношення цін ресурсів було вже давно вирішена, то з визначенням які елементи з безлічі варіантів треба використовувати для отримання найкращих результатів що задовольняють певні вимоги - поки є відчутні складності.

Ці складності мають під собою ті самі першопричини що і проблема бартеру у давні часи - людині складно порахувати усі фактори що впливають на ціну товару а відповідної системи за допомогою чого б таке рішення можна було б сильно спростити - просто не існує. Дана дисертація ставить собі за мету описати таку систему, за допомогою якої людина матиме змогу оперувати високорівневими сутностями такими як SLA, швидкість затримки, вартість системи, надійність системи і т.д. не витрачаючи час на вивчення існуючих низькорівневих сутностей, можливостей їх оптимізації і проектування необхідної системи.

Оптимізатор конфігурації розподілених інформаційних систем, за допомогою якого можна вирішувати наведені вище проблеми, має відповідати наступним **функціональним вимогам**:

- Можливість використання системи для вирішення задач, які важко або неможливо деталізувати на початку.
- Можливість пошуку альтернатив заявленому користувачем рішенню
- Можливість внесення існуючого рішення у систему для аналізу проблемних частин і способів їх вирішення.
- Можливість проведення коригування наявного оптимального рішення з мінімальними змінами після появи нових відомостей про компоненти системи.

Окрім цього система повинна задовольняти наступним **технічним вимогам**:

- Можливість внесення додаткових критеріїв для порівняння.
- Розширюваність системи. Кожен користувач повинен мати можливість описати наявні в нього ресурси простою декларативною мовою а в разі відсутності типу ресурсу в системі - додати.
- Доступ до мережі Internet для отримання даних про появу нових і зміну в існуючих сервісах. При цьому допускається використання будь-якої операційної системи та архітектури процесора на якій можна запустити Docker.
- Наявність зручного користувальницького інтерфейсу, використання якого не має вимагати високого технічного рівня від користувача.
- Відкритість програмного коду для змін, що дозволятиме залучати сторонніх розробників на безоплатній основі покращувати якість продукту,
- Обробка даних в тій системі де буде запущено обчислення, для запобігання пересилання конфіденційної інформації. Надсилатись має лише анонімізована статистична інформація що є важливою для

покращення роботи системи, і тільки в тому разі, якщо користувач не забороняє таке надсилання.

Перш ніж розібратись чи існують такі системи в сучасному світі, варто дати визначення Розподіленій інформаційній системі, розібратися в її складових і вимогах які до них висуваються.

1.1. Розподілена інформаційна система

Розподілена інформаційна система (РІС) - це сукупність технічних, програмних та інших засобів поєднаних структурно і функціонально для забезпечення одного чи декількох видів інформаційних процесів та надання інформаційних послуг. Сучасним РІС притаманні ієрархічність, функціональна розподіленість, високий ступінь розпаралелювання ресурсів (обслуговування, логіки, програмного та апаратного забезпечення, телекомунікацій, і практично повна відсутність централізованого управління [2].

Будь-яка РІС складається з цілого ряду програмних та апаратних комплексів. Розглянемо їх основні види.

1.1.1. Апаратна складова

Розрізняють три типи інфраструктури за типом доступу та підтримки апаратної складової:

- Локальна інфраструктура (On premise)
- Гібридна інфраструктура
- Хмарна інфраструктура (IaaS)

Локальна інфраструктура

Плюси: повний доступ; можливість поліпшити систему на апаратному рівні; дані не виходять з внутрішньої мережі компанії

Мінуси: відносно дорого на малих та середніх об’ємах; потребує обслуговування; додаткова точка відмови; масштабування займає багато часу

Хмарна інфраструктура / Infrastructure as a Service

Плюси: необмежені можливості для масштабування; інфраструктуру можна описати кодом; можна швидко підняти ідентичну копію; за працеспроможність інфраструктури відповідає провайдер.

Мінуси: піднімати ресурси занадто легко - залишаються «висячі ресурси» що збільшує кінцевий рахунок; апаратна оптимізація недоступна або сильно обмежена; різні юридичні аспекти.

Найвідоміші IaaS провайдери: Amazon Web Services, Microsoft Azure, Google Cloud Platform, IBM, Oracle, Alibaba Cloud.

Хмарна інфраструктура ділиться на 2 типи за кількістю використовуваних провайдерів: **cloud** або **multi-cloud**. Перший дозволяє користувачу отримати найбільшу вигоду від використання супутніх сервісів провайдера, другий - орієнтований на більшу стабільність роботи і вимоги покупців у B2B.

Гібридна інфраструктура

Покликана об’єднати плюси і мінімізувати мінуси локальної і хмарної інфраструктур. Також використовується як перехідний етап при покроковій міграції з одного типу інфраструктури в інший.

Висновок по АС

IaaS легші в використані і швидше масштабуються, тому для користувачів що не мають власних датацентрів економічно вигідно використовувати саме IaaS. Натомість для користувачів що вже мають локальну інфраструктуру вигідно її максимально утилізувати, а в разі сильної нестачі власних ресурсів потрібно будувати гібридну інфраструктуру з врахуванням вартості підтримки гібридної

інфраструктури та накладувані обмеження, або ж повністю мігрувати до хмарних провайдерів [3] [4].

1.1.2. Програмна складова

На програмне забезпечення (ПЗ) що працюватиме в PIC накладається цілий ряд архітектурних вимог, без виконання яких доцільність використання PIC зводиться до нуля. Далі розглянемо що являє з себе архітектура ПЗ, базові вимоги як до звичайного ПЗ, так і для ПЗ в PIC, що уточнюють перші.

1.1.2.1. Архітектура програмного забезпечення в циклі розробки

Архітектура програмного забезпечення має багато визначень, таких як:

“Найвищий рівень розбиття системи на частини; рішення, які важко змінити; наявність декількох архітектур у системі; те що може значущо змінити архітектуру протягом життя системи; і, врешті-решт, архітектура зводиться до важливих речей.” [5].

“Архітектура програмного забезпечення для програмної або обчислювальної системи - це структура або структури системи, що складаються з елементів програмного забезпечення, видимих зовнішніх властивостей цих елементів та відносин між ними. Архітектура має відношення до видимої сторони інтерфейсів; приховані деталі елементів — деталі, що пов'язані виключно з внутрішньою реалізацією — не є архітектурними.” [6]

Проте вони усі зводяться до наступного: архітектура програмного забезпечення — це процес визначення структурованого рішення, що відповідає всім технічним та операційним вимогам, при оптимізації спільних атрибутів якості, таких як продуктивність, безпека та керованість. Вона включає низку рішень, що базуються на різноманітних

факторах, і кожне з цих рішень може мати значний вплив на якість, продуктивність, ремонтопридатність та загальний успіх програми.

Архітектурним проектуванням називають перший етап процесу проектування, на якому визначаються підсистеми, а також структура управління і взаємодії систем.

Підсистема — це система, операції якої не залежать від сервісів, що надаються іншими підсистемами. Підсистеми складаються з модулів — системних компонентів, що надають один або кілька сервісів для інших модулів. Кожна підсистема може представляти чорний ящик з відомою функціональністю і інтерфейсом.

Для опису і налаштувань модулів використовуються конфігурації — форми інструкцій для засобів розгортання архітектури.

1.1.2.2. Вимоги, достатні для визначення архітектури програмного забезпечення

Моделі архітектури можуть залежати від функціональних, експлуатаційних та інших вимог що разом звуться нефункціональні вимоги [7]. Найважливіші з них:

- Ефективність. За критичні операції відповідає якомога менше підсистем, тобто використовується крупномодульна архітектура.
- Захищеність. Багаторівнева архітектура системи, найбільш критичні елементи захищені на нижньому рівні.
- Надійність. Включаються явно зайві компоненти, які можна змінювати не перериваючи роботу системи.
- Зручність супроводу. Архітектура складається з дрібних компонентів, які можна легко адаптувати під вимоги предметної області.
- Безпека. За всі операції, що впливають на безпеку, має відповідати якнайменше підсистем.

- Вартість. Сумарна ціна розробки не має перевищувати виділений на неї бюджет.
- Вартість супроводу. Ціна заміни елементу чи його поліпшення не має бути критичною.

Вимоги до РІС включають у себе вимоги до ПЗ і розширюють їх наступним:

- Ефективність. Системи обробки/видачі результату мають знаходитись якомога ближче до кінцевого користувача і одна до одного - для збільшення швидкодії.
- Захищеність. Ізоляція критичних частин не тільки на програмному, але й на фізичному рівні. Запуск критичних частин окремо від усіх інших систем.
- Надійність. Кожен програмний компонент має запускатися мінімум у 2-х екземплярах у різних датацентрах.

З чого випливає необхідність проектування критичних ПЗ таким чином, щоб інформація про вхідні параметри запиту до сервісу зберігалась ззовні; інформація про запити мала декілька станів (Новий, В роботі, Виконано); запит не був позначений як «виконаний» до отримання підтвердження про виконання; сервіс не змінював стан бази тощо в процесі обробки запиту до моменту виконання всіх операцій над даними; база розміщувалась окремо від ПЗ; обов'язковими стають реплікації типу master-slave, де запити на читання відправляються тільки на slave; з'являється балансувальник навантаження на сервіси тощо [8].

1.2. Методи інтеграції компонентів в РІС

Кожен сервіс складається з одного чи декількох компонентів. Для того щоб компоненти розуміли як взаємодіяти між собою, вони мають підтримувати деякий інтерфейс, з яким можуть спілкуватися інші

компоненти. Тут варто зазначити, що РІС за методами взаємодії діляться на жорстко зв’язані і слабозв’язані.

До жорстко зв’язаних відносяться системи де компоненти спілкуються на мережевому чи транспортному рівнях OSI, наприклад, mesh-мережа. До слабозвязаних відносяться системи, де компоненти спілкуються на прикладному рівні за допомогою таких засобів як прикладний програмний інтерфейс (API).

Будувати жорстко зв'язані РІС не жертвуючи при цьому надійністю чи вартістю системи на малих і середніх об’ємах досить складно, тому ми їх розглядати не будемо. Обмовимось тільки що вони дозволяють отримати найбільш можливий рівень швидкодії за рахунок зменшення накладних витрат на спілкуванн різних компонентів між собою.

1.2.1. Поняття інтерфейсу і уніфікація конфігурацій в РІС

Інтерфейс являє з себе сукупність засобів, через яку два або більше окремих компонентів комп'ютерної системи обмінюються інформацією. Обмін може здійснюватися між програмним забезпеченням, комп'ютерним обладнанням, периферійними пристроями, людьми та їх комбінаціями [9].

Для збільшення зручності супроводу і зменшення вартості розробки, прийнято уніфіковувати інтерфейси що зустрічаються у системі. Прикладами такої уніфікації може виступити будь-який опис специфікації, наприклад Simple Object Access Protocol (SOAP). SOAP Являє з себе протокол обміну структурованими повідомленнями на базі XML. Наразі він виступає в якості вимог до того як має формуватися запит/відповідь в XML-форматі, розширюючи можливості біль давньої специфікації XML-RPC.

Іншим прикладом специфікації до інтерфейсів виступає RESTful API. REST розшифровується як «передача репрезентативного стану» і бере свої витоки з HTTP 1.0. Даний підхід розроблявся з прицілом на роботу

Інтернет-сервісів з передачею невеликих шматків даних у форматі XML, JSON чи HTTP і кожен такий запит має бути повністю автономним від інших [10]. Хоча REST був надзвичайно успішним, він має свої недоліки, які в останні роки стали очевидними через те що застосунки з REST зростають в розмірах і складності.

Тому продовжують з'являтися нові специфікації інтерфейсів які спрямовані на зменшення кількості запитів через кешування (GraphQL), зменшення кількості трафіку через обмін бінарими повідомленнями (gRPC) або ж збільшення читабельності запитів людиною (JSON-RPC) тощо.

1.3. Формування і перевірка бізнес вимог

1.3.1. Етапи розробки PIC

Розробка розподіленої інформаційної системи, як було вже сказано раніше, наразі мало чим відрізняється від розробки програмного забезпечення - лише накладає додаткові вимоги до розроблюваного ПЗ.

Організація бізнес процесу розробки ПЗ, такого як Rational Unified Process (скорочено - RUP) [11] передбачає наявність фаз що відповідають за ключові етапи створення програмного продукту. Згідно RUP, існує 4 фази. На початковій фазі відбувається оцінка системи, визначення вартості розробки та необхідного бюджету. На фазі уточнення відбувається пом'якшення ключових ризиків, робиться аналіз предметної області і проектується базова архітектура. На фазі конструювання відбувається розробка системи та її компонентів, можлива зміна архітектури для забезпечення кращих характеристик розроблюваної системи. Фаза впровадження є орієнтованою на покращення системи для користувачів і навчання останніх нею користуватись.

Серед іншого, RUP описує 6 основних інженерних дисциплін, що так чи інакше присутні у кожному проекті:

- Дисципліни бізнес-моделювання
- Дисципліни вимог
- Дисципліна аналізу та проектування
- Дисципліна реалізації
- Дисципліна тестування
- Дисципліна розгортання

Кожна з них відповідає за ключові фази розробки продукту і простягаються у часі крізь різні фази, що зображено на Рис.1.1.

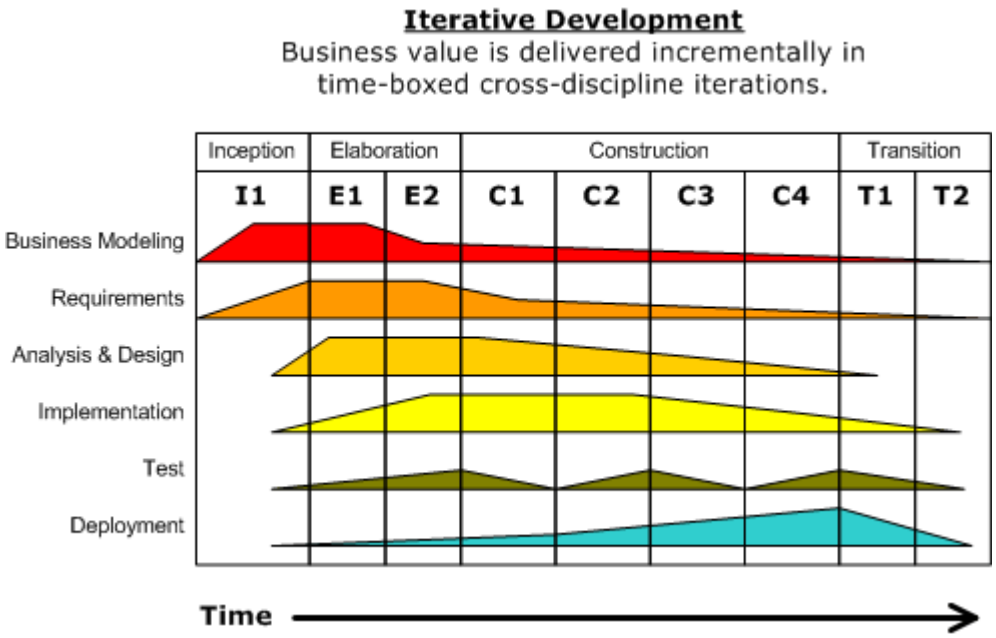

Рис.1.1. - Фази та дисципліни RUP [12].

Є декілька спільних характерних рис у кожної з дисциплін. Кожна з них має одну чи дві фази розробки, де їх роль найбільш значна. Чим пізніше виявляються якісь недоробки - тим дорожче коштує їх виправити. Особливо це помітно на етапах бізнес-моделювання і збору вимог до кінцевої системи. Їх найпростіше перевірити маючи прототипи кінцевої системи, що базується на поставлених вимогах.

1.3.2. Як відбувається проектування архітектури в RUP

У проекта є архітектор, або ж хтось, хто виконує його роль. Йому на вхід поступають вимоги у вигляді «розробити сервіс для %призначення%,

який би витримував пікове навантаження в 10млн одночасних користувачів». Архітектор розкладає це на вимоги, які в загальному вигляді можна описати як:

- Key-Value storage
- адмінка
- OAuth
- тощо

Далі відбувається збір з цих кубиків-вимог рішення, яке б задовольнило бізнес на основі наявного в архітектора досвіду роботи з тими чи іншими сервісами.

Остаточна композиція сервісів (Рис.1.2.) може бути надлишковою чи низькоефективною через брак знань про нові можливості тих чи інших сервісів, або відсутності досвіду з ними. Проблема у тому, що цього не видно, доки не запросити декількох сторонніх архітекторів зі схожим та абсолютно іншим стеком і порівняти їх рішення - так як кожен з них буде казати що його рішення виправдовується часом і наводити варті уваги розрахунки і пояснення чому кожен елемент стільки коштує, проте все це не відповідатиме на головні питання: чому такий стек і чи не дешевше було б реалізувати на альтернативному?

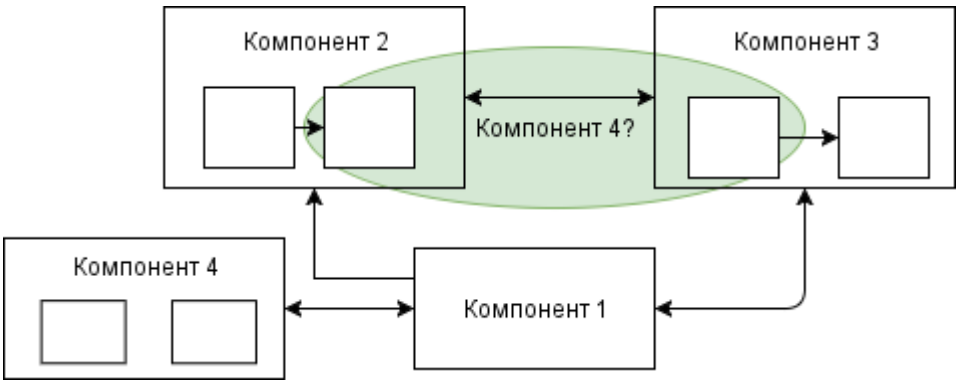

Рис.1.2. Приклад композиції сервісів

Недоліком такого підходу є занадто пізнє визначення архітектурного рішення яке буде лежати в основі реалізації. Зменшення часу на підготовку проектного рішення (артефакту) дозволяє отримати:

- Зниження технологічних ризиків (чим раніше - тим більше часу на нівелювання ризиків).
- Зниження вартості реалізації.
- Повторне використання підсистем.

1.3.3. Вибір конфігурації сервісів

Для зменшення часу на підготовку проектного рішення необхідно мати систему що міститиме знання про можливості сервісів і їх прямі і непрямі аналоги, і пропонуватиме кінцевому користувачу найбільш оптимальні конфігурації.

Наразі існують прості реалізації таких систем у сфері продажу товарів.

Наприклад, hotline.ua має набір “розумних” фільтрів відображених на Рис.1.3. [13]. Ці фільтри з себе являють часті юзкейси користувачів, при виборі яких автоматично вибираються більш конкретні фільтри, які, щоб не плутатись, будемо називати тегами.

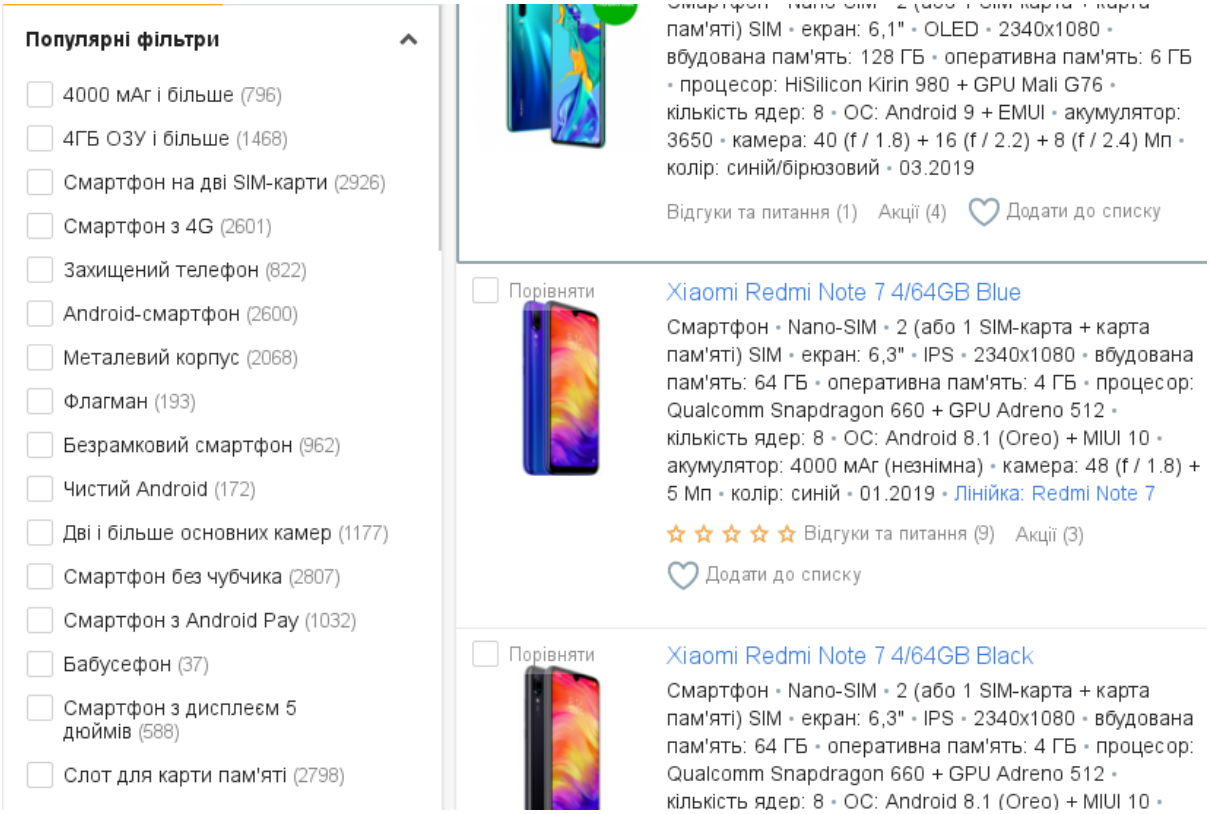

Рис.1.3. - Приклад набору «розумних» фільтрів на hotline.ua [13]

Юзкейс (з англ. Use Case) — це опис поведінки системи, як вона відповідає на зовнішні запити. Іншими словами, це різновид використання системи, де описується, «хто» і «що» може зробити системою.

Теги - це текстові чи інші поля, що використовуються як спосіб класифікації об’єктів в системі, за принципом багато-багато.

Можна сказати що питання сортування в масштабах інтернет-магазинів є вирішеним, хоча все ще далеко від правди. Однак відносно успішні приклади реалізації таких систем існують.

У той же час для опису PIC таких не існує, навіть в рамках єдиного провайдера. Що ж може допомогти у формуванні конфігурації для Розподілених інформаційних систем?

1.3.4. Аналіз існуючих рішень

1.3.4.1. Порівняння сервісів постачальників хмарних технологій

Найбільш серйозне порівняння всього лише 2-х хмарних провайдерів - Amazom Web Services і Microsoft Azure, яке було створене восени 2018 компанією Microsoft - це текстове порівняння у вигляді таблиці - Рис.1.4.[14]. Сама таблиця доступна на Github [15] під Creative Commons Public License, що дозволяє використовувати дані з таблиці. однак з деякими обмеженнями.

Також AWS 18 квітня 2019 року анонсувала підтримку незворотної міграції Azure VM в AWS EC2 в рамках сервісу AWS SMS [16] що дозволяє сподіватись на подальшу уніфікацію основних сервісів головних гравців і появу більш продвинутих інструментів міграції, що зпростить роботу у мультиклаудному середовищі і оптимізації ресурсів.

Marketplace

Area	AWS service	Azure service	Description
Marketplace	AWS Marketplace	Azure Marketplace	Easy-to-deploy and automatically configured third-party applications, including single virtual machine or multiple virtual machine solutions.

Compute

Area	AWS service	Azure service	Description
Virtual servers	Elastic Compute Cloud (EC2) Instances	Azure Virtual Machines	Virtual servers allow users to deploy, manage, and maintain OS and server software. Instance types provide combinations of CPU/RAM. Users pay for what they use with the flexibility to change sizes.
	Amazon Lightsail	Azure Marketplace	Azure Marketplace includes a selection of preconfigured VM images for various applications.
Container instances	EC2 Container Service (ECS)	Azure Container Service	Azure Container Instances is the fastest and simplest way to run a container in Azure, without having to provision any virtual machines or adopt a higher-level orchestration service.
	EC2 Container Registry	Azure Container Registry	Allows customers to store Docker formatted images. Used to create all types of container deployments on Azure.
Microservices / container orchestrators	Elastic Container Service for Kubernetes (EKS)	Azure Kubernetes Service (AKS)	Deploy orchestrated containerized applications with Kubernetes. Simplify monitoring and cluster management through auto upgrades and a built-in operations console.
		Service Fabric	A compute service that orchestrates and manages the execution, lifetime, and resilience of complex, inter-related code components that can be either stateless or stateful.

Рис.1.4. - Порівняння сервісів AWS і Azure [14]

Також існує проект що ставить на меті створити порівняння усіх сервісів великих постачальників хмарних технологій - CloudComparer [17], і, як видно з Рис.1.5., CloudComparer має досить поверхневе порівняння без опису що саме робить сервіс, як у порівнянні від Microsoft, однак, якщо їх поєднати, то можна отримати якість першого і покриття другого. CloudComparer розміщений на Github [18] під ліцензією MIT, що дозволяє використовувати цей проект без жодних обмежень.

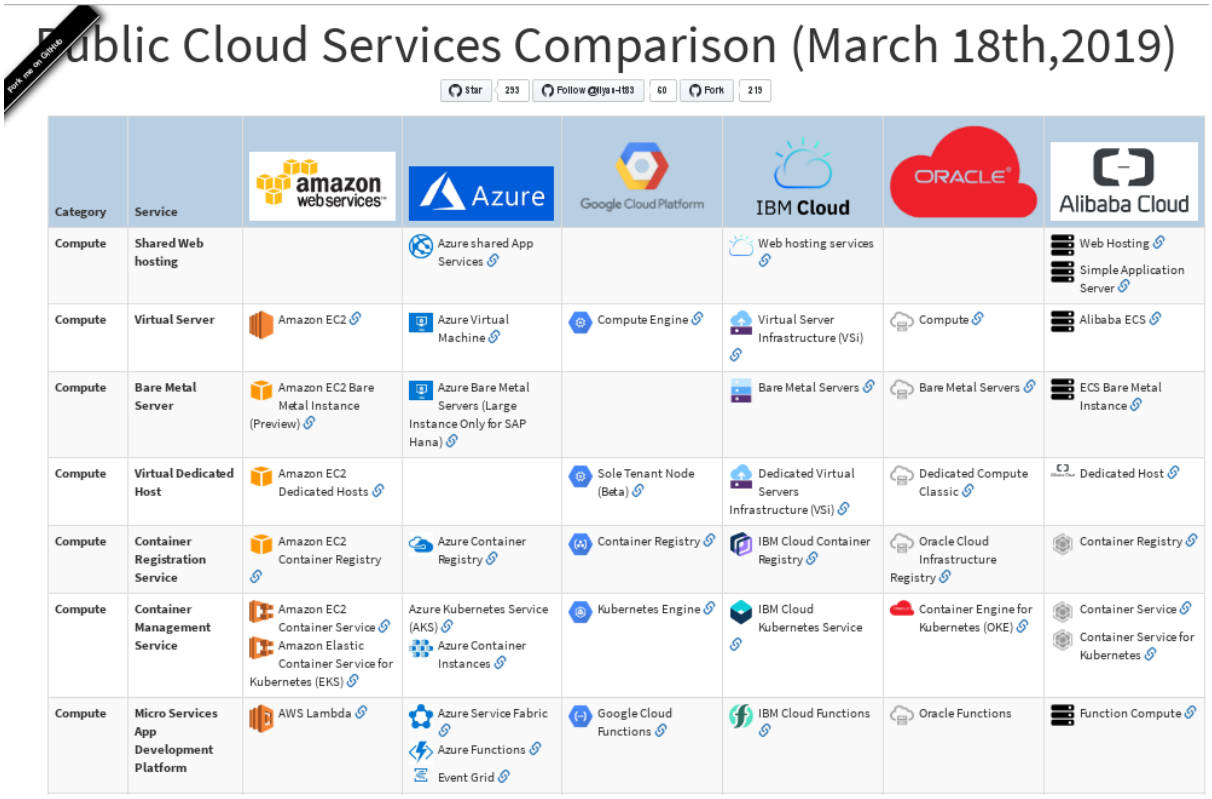

Рис.1.5. - Порівняння сервісів у CloudComparer [17]

1.3.4.2. Порівняння постачальників хмарних технологій AWS, Azure, GCP, IBM Cloud

Більш високорівневе порівняння на рівні сервісів і лімітів вже чотирьох провайдерів з можливістю експорту всіх даних у CSV форматі доступне на сайті Cloud Comparison Tool [19], Рис.1.6. Воно дозволяє на око визначити який з провайдерів краще підходить для вирішення тих чи інших задач, однак це не дозволяє будувати оптимальні системи, які в оптимальному варіанті можуть виявитись мультиклаудними [20].

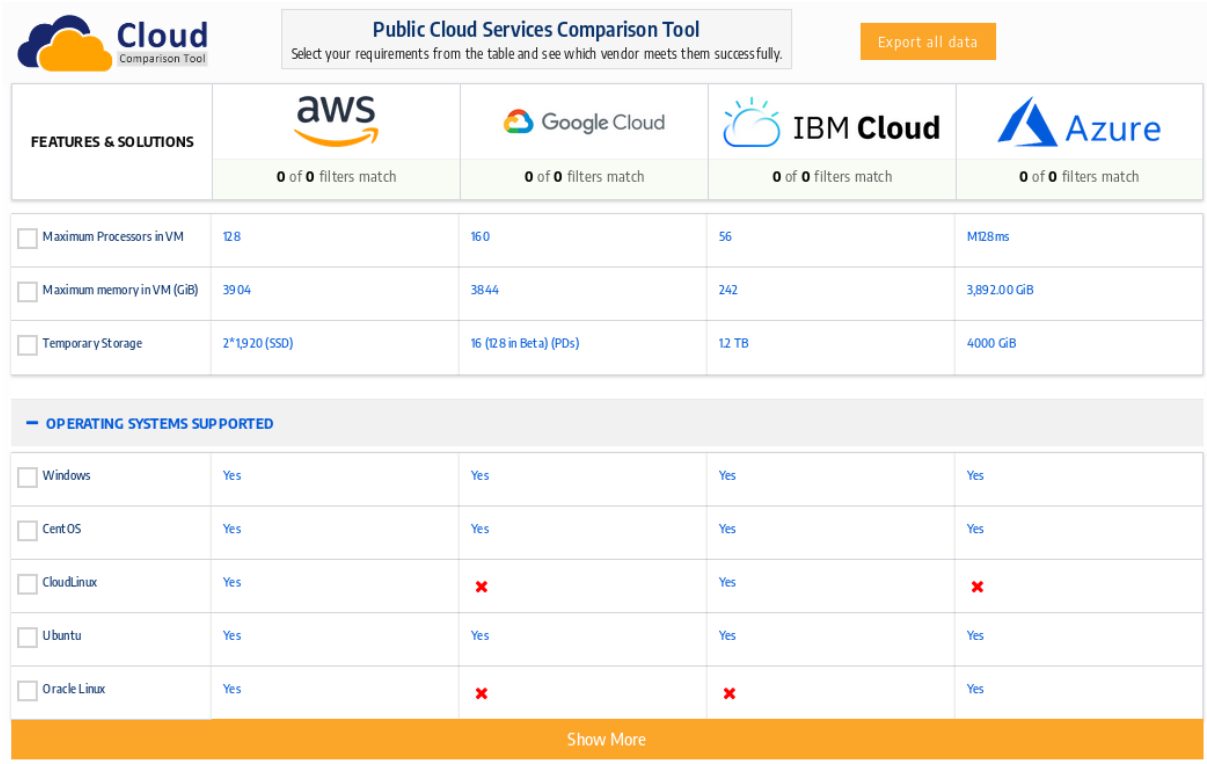

Рис.1.6. Сайт Cloud Comparison Tool [19]

1.3.4.3. Типові архітектурні рішення

В той же час у AWS, Azure і GCP є «типові рішення» з оптимальної побудови різних юзкейсів користувачів. Поки що їх не досить багато (так, станом на кінець квітня 2019 у Azure - 138 рішень, а у AWS - 43), та деякі юзкейси вони покривають а з часом їх має ставати все більше і більше [21][22][23]. Стартові сторінки сервісів зображені на Рис.1.7., Рис.1.8.. і Рис 1.9..

Частина з цих гайдів містить у собі чітку покрокову інструкцію як відтворити типове рішення, або Launch template, який розгортає необхідну інфраструктуру і підключає сервіси в декілька кліків. Іноді наявні відео і текст що повідомляють про переваги цього рішення перед усіма іншими. В гіршому випадку, наявний лише опис того що і де має бути аби рішення запрацювало, а не як це зробити або взагалі картинка зображення інфраструктури з мінімальними поясненнями, для прикладу - http://gcp.solutions/diagram/ba-dr-with-replication. Іншими словами, людина

не підготовлена навряд щось зможе зробити по більшості з цих гайдів, однак вже сама їх наявність - це великий здобуток, на якому можна будувати оптимізатор конфігурації розподілених інформаційних мереж.

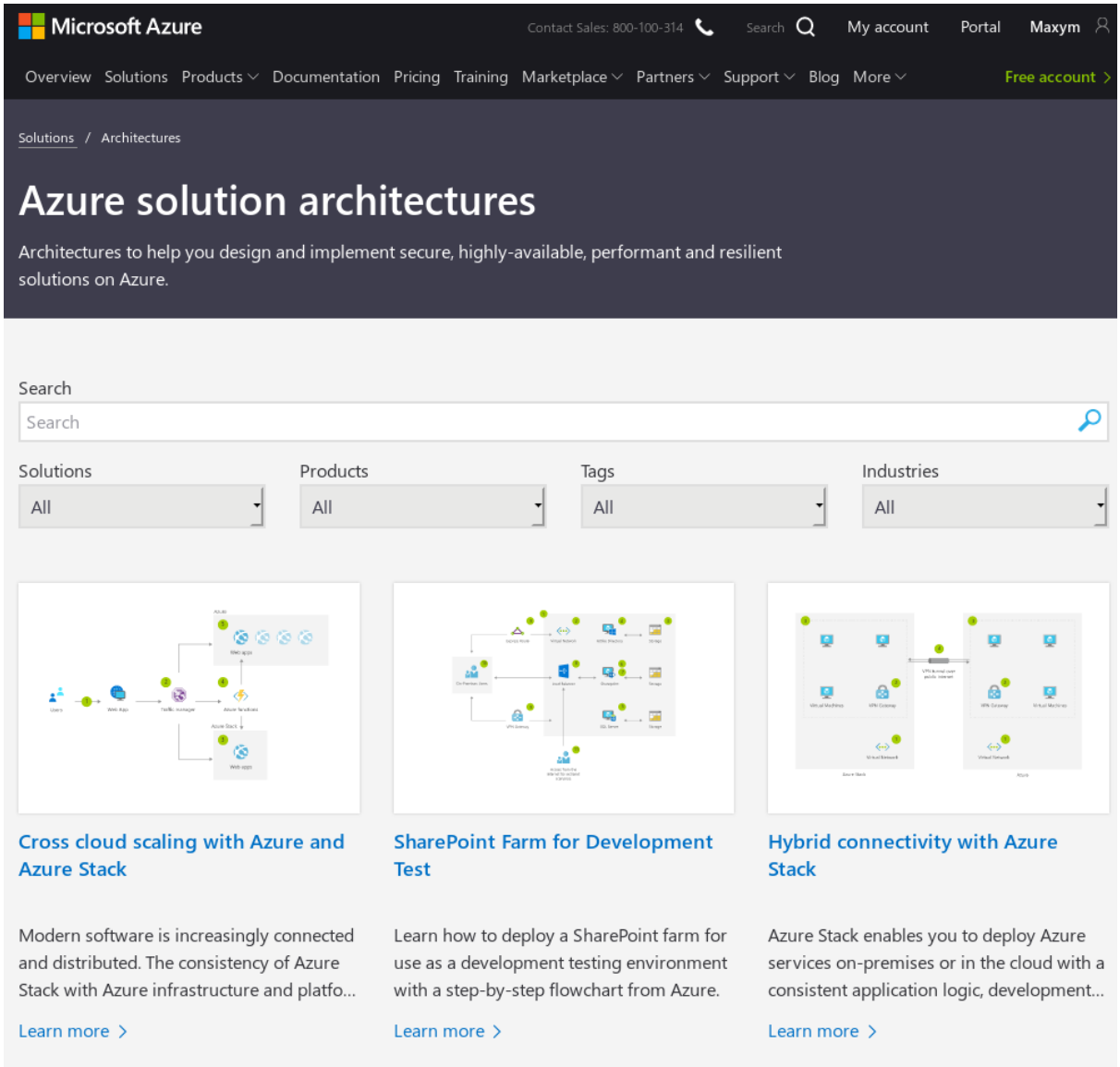

Рис.1.7. Стартова сторінка типових рішень на Azure [22]

Рис.1.8. Стартова сторінка типових рішень на AWS [21]

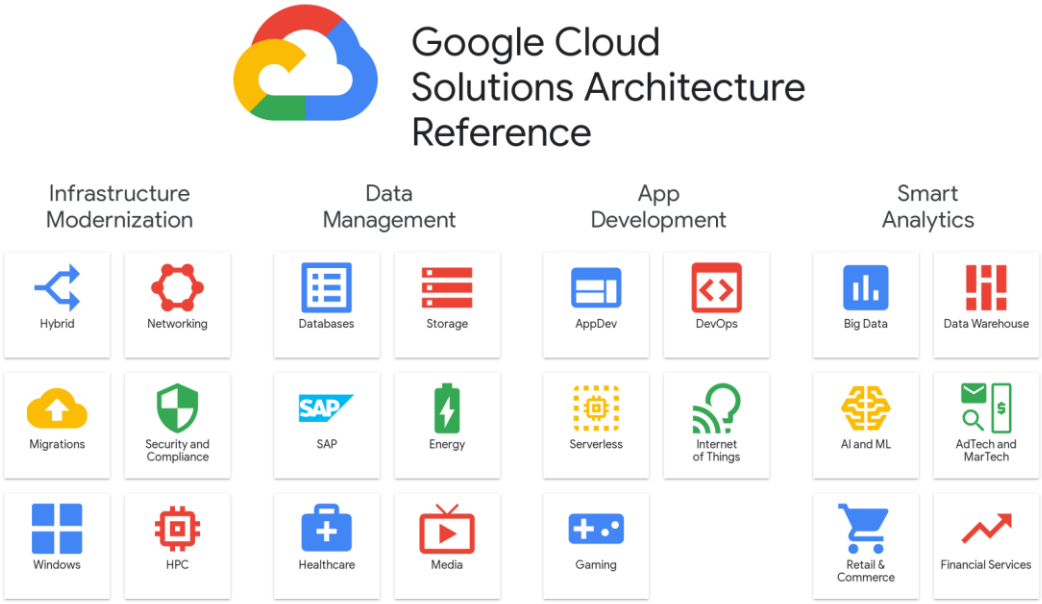

Рис.1.9. Стартова сторінка типових рішень на GCP [23]

1.3.4.4. Моніторингові системи, що виявляють проблеми

Існують платні застосунки і сервіси, які дозволяють знаходити проблеми в існуючій інфраструктурі і сигналізувати про покращення чи погіршення стану PIC після застосування тих чи інших змін. Для прикладу, таким функціоналом наділений NewRelic [24] - першокласне SaaS рішення для моніторингу, однак доволі дороге. Те саме пропонує інтеграція Datadog + CloudCheckr [25] - інше недешеве рішення для моніторингу.

У великий і середніх компаніях повсякчас є самописні розширення до систем моніторинга що реалізують той самий функціонал, однак такі розширення сильно заточені під специфіку компанії і не є загальнодоступними.

1.3.4.5. Порівняння існуючих рішень

Відповідно до функціональних і технічних вимог бажаної системи, переваги та недоліки існуючих рішеннь можна навести у Таблиці 1.1.

Таблиця 1.1 Порівняння існуючих рішень

Можливість	Назва системи							
	AWS to Azure services comparison	CloudComparer	Cloud Comparison Tool	AWS Solutions	Azure solution architectures	Google Cloud Solutions Architecture Reference	NewRelic	Datadog + CloudCheckr
Можливість використання системи для вирішення задач, які важко або неможливо деталізувати на початку.	-	-	-	+-	+-	+-	+	+
Можливість пошуку альтернатив рішення	+-	+-	+-	-	-	-	-	-
Можливість внесення існуючого рішення у систему	-	-	-	-	-	-	+	+
Можливість коригування наявного рішення з мінімальними змінами	-	-	-	-	-	-	не застосовно	не застосовно
Можливість внесення	-	-	+-	-	-	-		

додаткових критеріїв для порівняння.								
Розширюваність системи.	не застосовно	не застосовно	не застосовно	не застосовно	не застосовно	не застосовно	не застосовно	не застосовно
Отримання оновлень через Internet	+	+	+	+	+	+	+	+
Користувальницький інтерфейс зрозумілий не технічному користувачу	+-	-	-	+	+	+	+	+
Відкритість програмного коду	+-	+	+-	-	-	-	-	-
Обробка даних в тій системі де буде запущено обчислення	не застосовно	не застосовно	не застосовно	не застосовно	не застосовно	не застосовно	-	-

З аналізу наявних інформаційних ресурсів можна зробити висновок що наразі не існує автоматизованого рішення з оптимізації конфігурацій PIC, хоч деякі спроби в цьому напрямку і робляться. Жодних програмних засобів що відповідали б функціональним вимогам хоча б частково, виявити не вдалося.

Для можливості такої автоматизації, необхідно сформувати систему пошуку по тегам і юзкейсам спираючись на вже наявні описи сервісів і їх API. Для цього необхідно розробити форму опису, щоб на ранніх стадіях сформулювати опис системи, і в тій же формі можна було вибирати набір

покриття за описом функціональності, що і буде запропоновано у наступному розділі.

ВИСНОВКИ ДО РОЗДІЛУ 1

Сформульовано підхід до визначення архітектури на ранніх фазах життєвого циклу ПЗ виходячи з функціональних і не функціональних вимог.

Проаналізовано складові частини Розподіленої інформаційної системи, вимоги до них і як вони мають інтегруватися. Після кропіткого пошуку рішень що б задовольняли поставлені вимоги, довелось констатувати повну відсутність хоч якихось вартих уваги наробок з автоматизації. В той же час у світі шаленими темпами розвиваються різні підходи і методи що так чи інакше намагаються закрити відсутність рішень у цій сфері за допомогою описання загальних способів до вирішення частих юзкейсів. Однак це все ще вимагає досить значних технічних знань у даній сфері і є велике поле для помилок спричинених людським фактором.

Внаслідок цього необхідно розробити модель автоматизованого рішення що відповідає поставленим функціональним і технічним вимогам та реалізувати необхідні програмні засоби для демонстрації можливостей такої системи.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ОПТИМІЗАТОРА КОНФІГУРАЦІЙ РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНИХ СИСТЕМ

Оскільки необхідного автоматизованого рішення з оптимізації конфігурацій РІС, що відповідає функціональним і технічним вимогам, які наведені в першому розділі, не існує, приймається рішення про створення власного. Теоретичним підґрунтям для цього являються розглянуті складові частини РІС, вимоги до них, методи інтеграції компонентів і її етапи розробки.

Було виявлено що існують певні залежності між дисциплінами і фазами розробки, і чим пізніше виявляються недоробки тим більше коштує вартість їх виправлення. Через це дисципліни бізнес-моделювання, вимог і аналізу та проектування затягуються на досить значний термін що коштує значних коштів. Зменшення часу на ці дисципліни без втрати або з мінімальною втратою якості були б вельми доречні у сучасному динамічному світі, де половина нових компаній в США закриваються за 4 роки з моменту заснування [26], а для Індії середній термін життя стартапу складає всього лиш 11.5 місяців [27]. Компанії часто не мають часу на розробку якісного продукту, а поганеньке зараз вже нікому не продаси. В той же час існуючому бізнесу необхідно оптимізовувати і перетворювати наявні інформаційні системи для збільшення конкурентоспроможності.

Однак внаслідок відкритості світу, провайдерів РІС існує більше одного що робить заскладним для людини визначення оптимальних конфігурацій. В той же час, відсутній стандартизований формат опису конфігурацій, що стає на заваді у реалізації автоматизованого оптимізатора РІС. Визначимо вимоги і створимо його.

2.1. Вимоги до конфігурації PIC

Якщо зробити конфігурацію не розширювальною користувачами, то існує велика ймовірність що якийсь компонент у користувача буде не описаний в жодному з типів конфігурації і він не зможе його додати без участі розробника що супроводжує конфігурацію.

Тут виникає умова до людиночитабельності, бо якщо користувачу буде складно зрозуміти що в конфігурації за що відповідає, він не зможе сам її розширювати чи уточнювати.

Вимога до компактності є водночас наслідком вимоги до читабельності (чим менше конфігурація, тим легше її осягнути людині), так і з практичних цілей економії місця і обчислювальних ресурсів.

Однак такою формалізованою системою буде складно користуватися непідготовленим користувачам. Однак не передбачається що нові компоненти будуть додавати спеціалісти без необхідного технічного підґрунтя, тож на даному рівні така складність є допустимою. В той же час у нетехнічних спеціалістів будуть складності з формальним описом того що вони хочуть побудувати з причини нерозуміння того що саме з безлічі варіантів їм треба. Тому опис того ЩО має будуватись з формально описаних ресурсів, буде неформальним.

Звідси випливає необхідність поєднати неформальний опис кінцевого користувача з формальним описом конфігурацій. Дану задачу пропонується вирішити за допомогою множинної класифікації елементів, яку можна реалізувати у вигляді тегів. Також потрібна буде класифікація кожного елемента по типу використання

Підсумовуючи, конфігурація PIC має відповідати наступним вимогам:

- бути розширюваною
- бути людиночитабельною
- бути компактною
- мати підтримку множинної класифікації

На множинній класифікації варто зупинитись окремо і прояснити деякі аспекти з нею пов’язані.

2.1.1. Множинна класифікація

В даному випадку під множиною класифікацією мається 2 схожих за змістом і реалізацією та абсолютно різні за вирішенням підходи. Це багато-класова та багато-ярликова класифікації.

Багато-класова класифікація (англ. Multiclass classification, далі - БКК) - це проблема класифікації елементу в один з 3-х чи більше класів у такій сфері знань як машинне навчання. Часто вирішується зведенням до бінарної класифікації, де застосовуються такі стратегії як один-проти-всіх та один-проти-одного, опис яких не є необхідним у рамках даної роботи. З підходів і способів що застосовуються для вирішення цієї проблеми варто виділити Наївний баєсів класифікатор, Дерево ухвалення рішень, Метод k-найближчих сусідів та Метод опорних векторів що є найбільш простими і пристосованими для задоволення поставлених вимог [28]. Їх можна використати для автоматичної класифікації ресурсів описаних і внесення у їх формальну конфігурацією класу до якого типу даний ресурс належить (інстанс БД, інстанс, інстанс функції, балансувальник навантаження тощо).

Багато-ярликова класифікація (англ. Multi-label classification, далі - БЯК) - це проблема класифікації елементу, який може бути віднесений до декількох категорій водночас. Є узагальненням багато-класової класифікації, де кожен елемент може бути віднесений тільки до однієї категорії. Алгоритми що застосовуються для вирішення проблеми БКК, після незначної адаптації можуть бути використані і для БЯК [29]. Для визначення підходящих формальних елементів з неформального опису

вимог користувача, якраз найкраще підходить багато-ярликова класифікація.

Однак, навіть зі значним спрощенням по використанню схожих алгоритмів для класифікації великої кількості елементів системи, застосування машинного навчання для демонстрації можливостей системи є надто амбітною і ресурсозатратною ціллю, тож попервах можна обійтись розміткою даних звичайними людьми, а далі отриману базу можна буде використати для навчання нейронної мережі чи тестування реалізації одного з вищенаведених алгоритмів.

Отже, використання машинного навчання для пришвидшення процесу додавання нових елементів і покращення ефективності системи є бажаним, однак не обов'язковим і тому в рамках даної дисертації буде розглядатися лише з теоретичної точки зору.

2.2. Взаємодія користувача з системою

Як вже описувалось вище, за типом взаємодії користувачів можна поділити на:

1. Користувач який вносить нові формальні описи елементів у систему, далі - К1
2. Користувач що неформальною мовою описує вимоги до системи, далі - К2

В обов'язки К1 входить якомога точніше і повніше заповнення кожного поля опису нового елементу, а також, за можливості, описувати найкращі архітектурні рішення під конкретні юзекейси за допомогою нових і наявних елементів. Базова категоризація елементу за допомогою тегів і прив'язкої тегів під юзкейси є однією з вимог що накладаються на К1.

К2 може обрати зі списку існуючих юзкейсів підходящий і ознайомитись з варіантами реалізації. Також, він може ввести додаткові фільтри-обмеження чи спробувати пошукати оптимальний варіант за допомогою неформального опису вимог до системи. В разі відсутності підходящого юзкейсу, для покращення кінцевого результату, буде запропоновано вибрати більш конкретні параметри для пошуку і ваги кожного з фільтрів для отримання найбільш відповідної вимогам користувача архітектурної моделі.

2.3. Архітектура системи

Структура програмних засобів виглядає як три пов’язані модулі, що поділяються за функціоналом: препроцесор, оптимізатор, постпроцесор. Також варто виділити імпортерів, що є необов’язковими для функціонування системи, однак корисними для підтримки актуальності даних в автоматичному режимі з джерел третіх сторін. Взаємодія елементів зображена на Рис. 2.1.

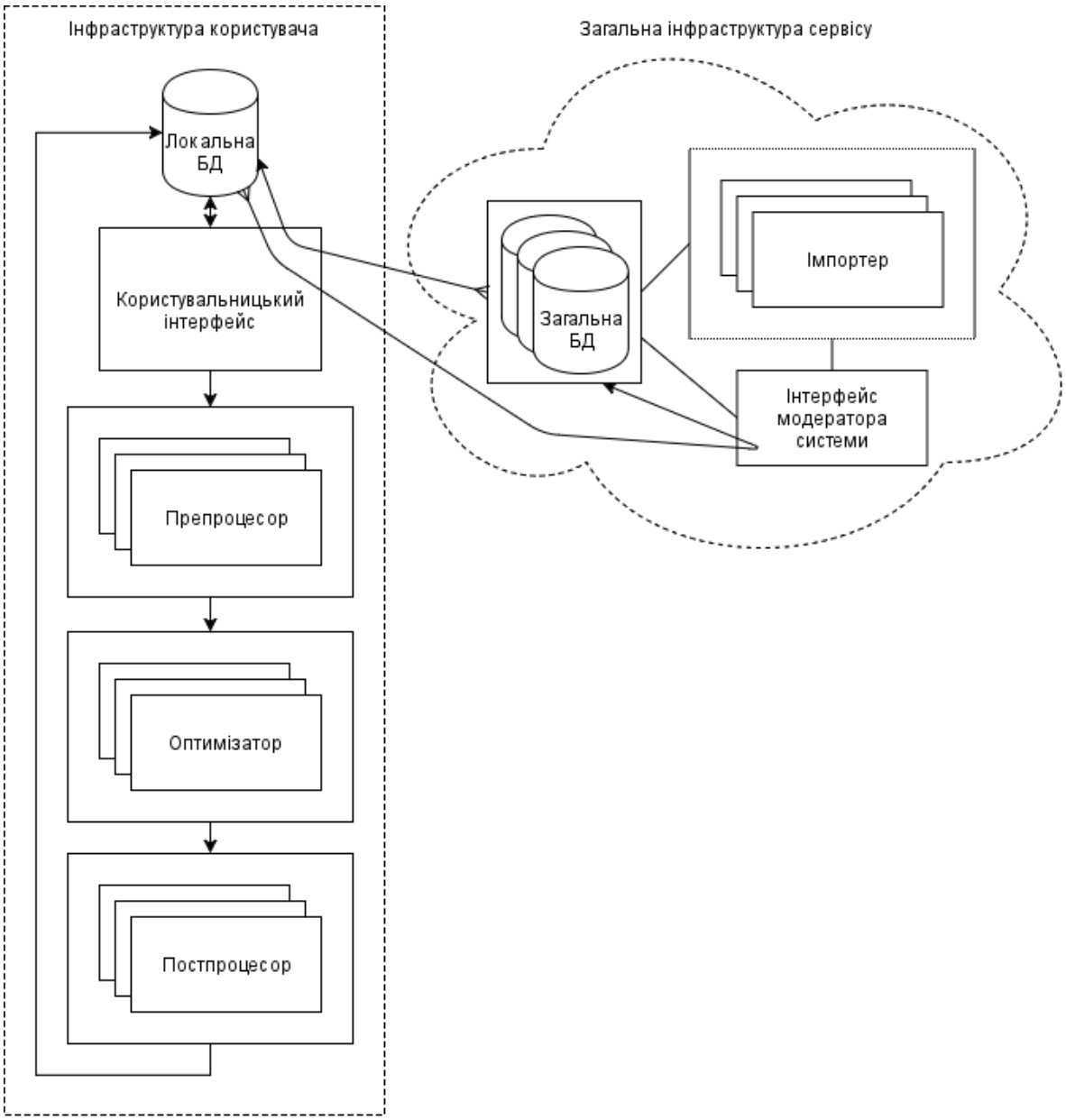

Рис.2.1. Елементи системи і їх взаємодія

Для створення єдиної точки комунікації у системі і для закладення в архітектуру вирішення задачі масштабування, буде використано RESTful API.

2.3.1. Препроцесор

Препроцесор відповідає за підготовку даних.

На вхід йому подаються побажання користувача до кінцевої PIC, з якої витягуються вимоги до типів необхідних елементів і всі наявні описи елементів що є в системі. Препроцесор починає пошук перетинів

неформального опису системи заданого користувачем і формального опису елементів. По закінченню пошуку, препроцесор видає список елементів що задовольняють вимогам користувача, відсікаючи усі елементи що чий клас не може застосовуватись для вирішення поставленої задачі.

Таким чином, результатом роботи препроцесора є набір всіх можливих комбінацій конфігурацій PIC що покривають вимоги до системи.

2.3.2. Оптимізатор

Оптимізатор відповідає за знаходження покриття і вирішення задачі оптимізації. При цьому оптимізатор буде зважати на існуючі типові архітектурні рішення, що має істотно зменшити ймовірність потрапляння неадекватних варіантів до кінцевого користувача.

На вхід приймає список усіх підходящих елементів що були визначені препроцесором, їх характеристики і ваги вимог до оптимізації системи.

Починається виконання задачі оптимізації PIC з оглядом на задані користувачем критерії і їх вагу. Задачі оптимізації і її аспектам буде присвячений весь наступний розділ, тож наразі не будем вдаватись у деталі. А от пошук покриття варто описати.

Покриття шукається за допомогою поєднання множини тегів і множини компонентів, при чому на даному етапі нас не цікавлять різниці у можливих конфігураціях кінцевих елементів що підходять під визначений клас елементів, достатньо лише визначити що в цьому класі є елементи що можуть задовольнити якусь з потреб системи. Це потрібно для відкидання класів елементів що ніяким чином не зможуть застосовуватись у даному юзкейсі, що призведе до формування списку класів елементів серед яких потрібно буде шукати оптимальні конфігурації.

2.3.3. Постпроцесор

Постпроцесор вибирає оптимальні параметри кожної з конфігурацій ґрунтуючись на вазі кожного з фільтрів, що були задані користувачем і порівнюються отримані результати.

На виході, за можливості отримати декілька схожих проектів архітектури, користувачу буде надана можливість порівняти характеристики кожного з варіантів і обрати той, що більше відповідає його цілям.

В момент вибору потрібного опису конфігурації, користувачу видається код що описує його інфраструктурні залежності, за допомогою якого користувач в автоматичному режимі зможе розгорнути систему будь-якої складності.

2.3.4. Імпортери

Імпортери є сервісами що в автоматичному режимі наповнюватимуть систему новими архітектурними рішеннями і елементами, що будуть підготовлені третіми сторонами яким можна довіряти. Вони не є обов'язковими, однак без них кількість необхідних людських ресурсів для підтримання бази елементів в актуальному стані буде зависокою.

2.3.4.1. Імпортери елементів

Умовно, імпортери елементів можна поділити на 2 групи. Перша - це імпортери що вказують на функціонально взаємозамінні або схожі сервіси в рамках одного провайдера чи між різними. Друга - імпортери що дістають всі можливі варіації характеристик сервісу. Також і перші, і другі можуть частково виконувати задачу класифікації сервісу і визначення його сфери застосування і можливості комбінації з іншими елементами.

У попередньому розділі були розглянуті рішення для порівняння сервісів і функціоналу постачальників хмарних технологій. Розглянемо їх детальніше і те як їх можна застосувати.

Порівняння сервісів від Microsoft [14] доступне на Github в Markdown таблицях. Не найкраще з можливих представлень даних, однак витягнути інформацію з такого подання є досить простою задачею.

З CloudComparer [17], що порівнює більше постачальників хмарних технологій, ще простіше - дані предстають у вигляді нормалізованої таблиці в форматі XLSX [30], а для роботи з цим форматом вже існує цілий спектр бібліотек [31].

Cloud Comparison Tool [19] порівнює функціональні можливості і дозволяє експортувати всі дані у форматі CSV, з яким теж не має жодних проблем.

2.3.4.2. Імпортери архітектурних рішень

Інший тип імпортерів - імпортери існуючі типових рішень. Типові рішення використовуються на етапі оптимізації для відсіювання занадто заплутаних архітектурних рішень і вважаються перевіреними оптимальними рішеннями, що застосування яких має зменшити час витрачений на оптимізацію.

Типові архітектурні рішення є найбільш повними і структурованими для AWS [21], Azure [22] і GCP [23]. Однак у той же час більшість з них не піддаються парсингу через неструктурованість тексту або ж й повну відсутність будь-чого крім схематичного зображення інфраструктури. Також більшість мають істотний недолік у вигляді відсутності опису необхідної інфраструктури у вигляді коду чи списку використовуваних елементів і формалізованого опису їх взаємозв'язків. Це все сильно ускладнює розробку автоматичних імпортерів архітектурних рішень.

2.4. Користувальницький інтерфейс системи

При розробці програмного забезпечення, до коду застосунку виникає необхідність підключати зовнішні залежності у вигляді інших програмних

модулів, системи зовнішнього зберігання (такі як бази даних, key-value storages або ж хмарні сховища) або ж запускати декілька копій одного застосунку паралельно. При цьому, у розробника можуть бути відсутні необхідні потужності що дозволяють розгорнути усю систему локально, або ж розгортання займатиме відчутний проміжок часу і тому простіше тримати частину або всю інфраструктуру запущеною у зовнішніх провайдерів.

Загальний взаємозв'язок між тегами і компонентами зображено на Рис. 2.2.

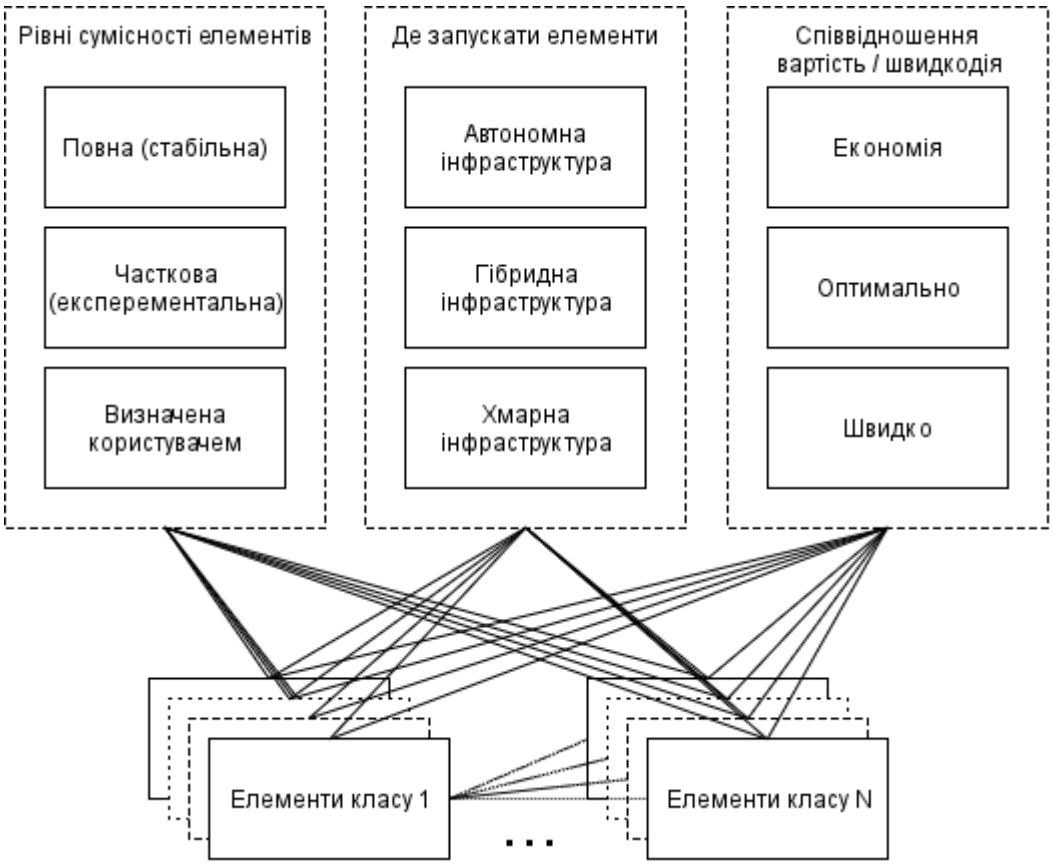

Рис.2.2. Узагальнене зображення взаємозв'язків між тегами, фільтрами і елементами

До програмного інтерфейсу підключаються задані користувачем K2 компоненти. Кожен з компонентів матиме один з трьох рівнів сумісності з іншими компонентами:

- Повна (стабільна)

- Часткова (експериментальна)
- Визначена користувачем

Повна - означає повну сумісність з усіма офіційно підтримуваними компонентами.

Часткова - означає що компонент працює у деяких конфігураціях, проте гарантія роботи в інших конфігураціях не гарантується.

Визначена користувачем - конфігурація задана користувачем. Жодних гарантій працеспроможності компонента не надається.

В залежності від вимог до інфраструктури та потужностей комп'ютера користувача, буде можливість вибрати тип розгортання для кожного з сервісів.

А саме:

- Запустити локально
- Запустити на віддаленому ресурсі

Над цим рівнем буде побудована абстракція по вибору оптимальної конфігурації для користувача у вигляді "розумних" фільтрів:

Перша група:

- Автономна інфраструктура
- Гібридна інфраструктура
- Хмарна інфраструктура

Друга група:

- Економія
- Оптимально
- Швидко

Автономна інфраструктура – усі сервіси запускаються локально на машині користувача.

Гібридна інфраструктура – сервіси запускаються як локально, так і у хмарі.

Хмарна інфраструктура – усі сервіси запускаються у хмарі.

Економія – сервіси запускаються з мінімальними параметрами інфраструктури. Пріоритети: 1 – ціна, 2 – швидкодія. За умови доступності безкоштовних хмарних рішень, що дозволяють запустити деякі сервіси з оптимальними параметрами – відбувається запуск з оптимальними параметрами.

Оптимально – сервіси запускаються з оптимальними параметрами інфраструктури. Пріоритети: 1 – ціна, 2 – швидкодія.

Швидко – сервіси запускаються в інфраструктурі орієнтованій на швидкодію. Пріоритети: 1 – швидкодія, 2 – ціна.

В рамках вибору конфігурацій виконуватиметься оптимізаційна задача.

Для цього у кожного сервісу будуть масиви метаданих, які відповідатимуть, за параметри запуску сервісу.

У параметри запуску входять наступні сутності:

- економні, оптимальні та орієнтовані на швидкодію налаштування
- залежності від інших сервісів
- хмарні та локальні провайдери

ВИСНОВКИ ДО РОЗДІЛУ 2

Було проаналізовано і створено список вимог до конфігурації розподіленої інформаційної системи, деякі з яких виявились незвичними. Так стало зрозуміло що деякі задачі вирішуються за допомогою машинного навчання що не входило в задачі цієї дисертації, однак довелось розглянути і ці можливості.

Одним з необхідних умов успішної реалізації відповідних програмних засобів є формування реєстру метаданих доступних для використання підсистем, на основі яких повинні визначатися альтернативні конфігурації.

Далі було описано взаємодію користувачів з системою, які можливості є у користувачів і які накладаються на них вимоги.

Також висвітлюється архітектура системи, те як система буде працювати з даними, ключові частини системи і те як вони між собою взаємодіють. Порушена тема автоматизації наповнення даними бази системи, однак через слабку підготовку даних що їх надають треті сторони, це не завжди можна зробити і доведеться або підготовлювати дані, або вручну вносити більшу частину наявних типових архітектурних рішень.

Оптимізація PIC в цьому розділі не висвітлюється у зв’язку з тим що це досить обширна тема і тому її висвітлення відбуватиметься у наступному розділі.

РОЗДІЛ 3
ПІДХОДИ ДО ОПТИМІЗАЦІЇ РОЗПОДІЛЕНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Ось і дісталися ключової теми диплому - оптимізації РІС.

Що ми вже на цей момент знаємо?

На вхід оптимізатору подаються елементи підходящих для вирішення юзкейса класів а також критерії оптимізації і їх вага. А на виході ми очікуємо побачити набір конфігурацій що максимально відповідають заданим критеріям. Звучить зрозуміло, однак проблеми виникають при реалізації деталей. Розкриємо же їх.

3.1. Завдання оптимізації

Завдання оптимізації конфігурації ставиться на безлічі альтернативних конфігурацій, отриманих на безлічі доступних підсистем що покривають функіональні вимоги. f - критерій оптимізації, що обчислюється на основі оцінок нефункіціональних вимог.

$$f \subseteq \textit{Ресурси} \times \textit{Вимоги}$$

При цьому нефункціональні вимоги є аргументами критеріїв оптимізації.

Завдання оптимізації базуються на переліку встановлюваних вимог до системи і їх пріоритетності.

У кожного Ресурсу є конфігураційні метадані, що описують усі можливі характеристики і вимоги до використання даного Ресурсу. За допомогою них можна вираховувати усі можливі композиції, відкинути рішення що не задовольняють вимоги і надати кінцевому користувачу найбільш підходящі конфігурації систем, тим самим виконуючи задачу оптимізації і зменшуючи час на підготовку проектного рішення. Задача набуває наступного вигляду:

$$\subseteq \qquad \in$$

$$f \quad Ресурси \ \mathbf{x} \ Вимоги \ \mathbf{x} \ R, \quad R \quad [0,1]$$

Де R - дійсне число від 0 до 1 включно і базується на атрибутах використаних у рішенні Ресурсів.

Відбуватиметься оптимізація під один з параметрів, наприклад, мінімальна кількість використовуваних ресурсів або мінімальна вартість:

$$\mathbf{F}(покриття) \subseteq \rightarrow min \mid \mathbf{F}(вартість) \rightarrow min$$

$$f \quad Ресурси \ \mathbf{x} \ Вимоги$$

Дану концепцію можна представити у вигляді таблиці, приклад якої наведений на Рис. 3.1., де у стовпцях описані Ресурси, а у рядках - Вимоги, які може задовольнити Ресурс.

Resouses → Requerements ↓	Local MongoDB	Local MySQL	Local MariaDB	Local Redis	AWS Elasticache	AWS RDS
Cost $/day	0	0	0	0	0.5	0.2
Storage	1	1	1	1	0	0.1
Memory	0.2	0.2	0.15	1	0	0

Рис. 3.1. Візуальне представлення пошуку оптимальної композиції

3.2. Існуючі алгоритми

Існує безліч алгоритмів що тим чи іншим способом можуть вирішити частину з поставлених задач. Дослідимо які алгоритми можуть стати в нагоді.

3.2.1. ABC-аналіз

ABC-аналіз — метод, який дозволяє класифікувати бізнес-ресурси фірми залежно від їхньої значущості. В основі класифікації лежить принцип Парето.

Відносно ABC-аналізу правило Парето виглядає таким чином: надійний контроль 20% позицій дозволяє на 80% контролювати систему, приклад на Рис.3.2. У бізнесі принцип ABC-аналізу та принцип Парето використовуються найчастіше у логістиці для управління товарними

запасами: стосовно запасів сировини, комплектуючих, постачальників, клієнтів тощо.

Наприклад, здійснивши ранжування запасів підприємства за значимістю, ми можемо визначити категорію «А» (скажімо, 10% запасів, вартість яких становить 70% усіх видатків), категорію «В» (20% запасів, які становлять 20% загальних видатків) і категорію «С» (решту запасів, до яких входять до 70% номенклатури, які займають, десь близько 10% усіх витрат). Таким чином, керівництво логістичного напрямку повинно сконцентрувати увагу на управлінні запасами категорії «А». Відносно запасів групи «В» контроль може бути періодичним, щодо категорії «С» — ще рідшим.

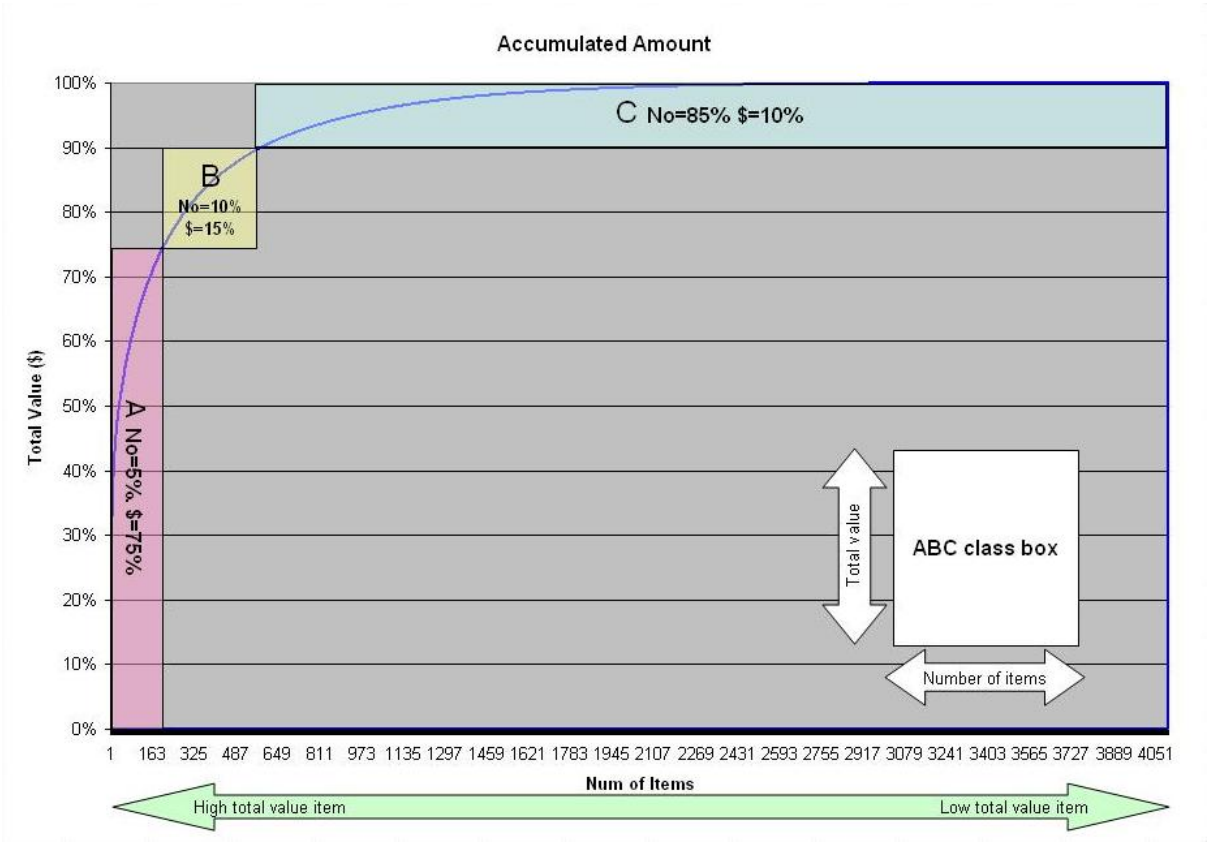

Рис.3.2. Приклад застосування зваженої операції на основі класу ABC [32]

ABC-аналіз можна використати для знаходження елементів, що сукупно покривають більшу частину вимог.

Також існує **Інтегрований ABC/FMR/XYZ/VED-аналіз** ABC-аналіз, який здійснюється одразу з іншими аналізами та формує матрицю інтегрованого ABC/FMR/XYZ/VED-аналізу. Формування такої матриці вручну є досить важким процесом, тому існують системи для його автоматизації (Hortor і т.д.) [33]

3.2.2. Класичні методи мінімізації

Мінімізація булевих функцій вручну за допомогою класичних карт Карно є трудомістким, стомлюючим і схильним до помилок процесом. Він не підходить для більш ніж шести вхідних змінних і практичний лише для чотирьох змінних [34] Більш того, цей метод не піддається автоматизації у вигляді комп'ютерної програми.

Першим альтернативним методом, що став популярним, був табличний метод, розроблений Віллардом Куайном і Едвардом МакКласкі. Починаючи з таблиці істинності для набору логічних функцій, поєднуючи мінтерми, для яких активні функції (ON-cover) або для яких значення функції не має значення (покриття Don't-Care або DC-cover) складається безліч простих імплікантів. Нарешті, дотримується систематична процедура, щоб знайти найменший набір основних імплікантів, з якими можна реалізувати вихідні функції.

Хоча **метод Куайна — Мак-Класкі** дуже добре підходить для реалізації в комп'ютерній програмі, результат все ще далекий від оптимального з точки зору часу обробки та використання пам'яті - додавання змінної до функції приблизно вдвічі збільшить їх, оскільки довжина таблиці істинності зростає експоненціально з числом змінних. Аналогічна проблема виникає при збільшенні числа вихідних функцій комбінаційного функціонального блоку. Як результат, метод Куайна —

Мак-Класкі практичний тільки для функцій з обмеженим числом вхідних змінних і вихідних функцій.

Функції з великою кількістю змінних мають бути мінімізовані з допомогою потенційно не оптимального евристичного алгоритму. На сьогодні евристичний алгоритм мінімізації Еспресо є фактичним світовим стандартом [35], розглянемо і його.

3.2.3. Алгоритм ESPRESSO

Радикально інший підхід до цього питання використовується в алгоритмі ESPRESSO, розробленому Брайтоном в Каліфорнійському університеті, Берклі. Замість того, щоб розширювати логічну функцію в мінтерми, програма маніпулює «кубами», що представляють терміни продукту в ON-, DC- і OFF-покриттях ітеративно. Хоча результат мінімізації не є глобальним мінімумом гарантовано, на практиці видається дуже наближений результат, і рішення завжди виходить вільним від надмірності. У порівнянні з іншими методами, цей по суті є найбільш ефективним, зменшуючи використання пам'яті та час обчислення на кілька порядків. Його назва відображає спосіб миттєвого приготування свіжої кави. Навряд чи існує обмеження щодо кількості змінних, вихідних функцій та термінів продукту комбінаційного блоку функцій. Загалом, легко обробляються десятки змінних з десятками вихідних функцій.

Вхідними даними для ESPRESSO виступає таблиця функцій бажаної функціональності; результатом є мінімізована таблиця, що описує або ON-покриття або OFF-покриття функції, залежно від обраних варіантів. За замовчуванням терміни продукту будуть спільними для якомога більшої кількісті функцій виводу, але програмі можна доручити обробляти кожну з вихідних функцій окремо. Це дає можливість ефективної реалізації в дворівневих логічних масивах, таких як PLA (Programmable Logic Array) або PAL (Programmable Array Logic).

Алгоритм ESPRESSO, зображений на Рис.3.3. виявився настільки успішним, що він був включений як стандартний крок мінімізації логічних функцій у практично будь-який сучасний інструмент синтезу логіки. Для реалізації функції в багаторівневій логіці результат мінімізації оптимізується шляхом факторизації і відображається на доступні базові логічні комірки в цільовій технології, незалежно від того, чи стосується це FPGA (Field Programmable Gate Array) або ASIC (Application Specific Integrated Circuit).

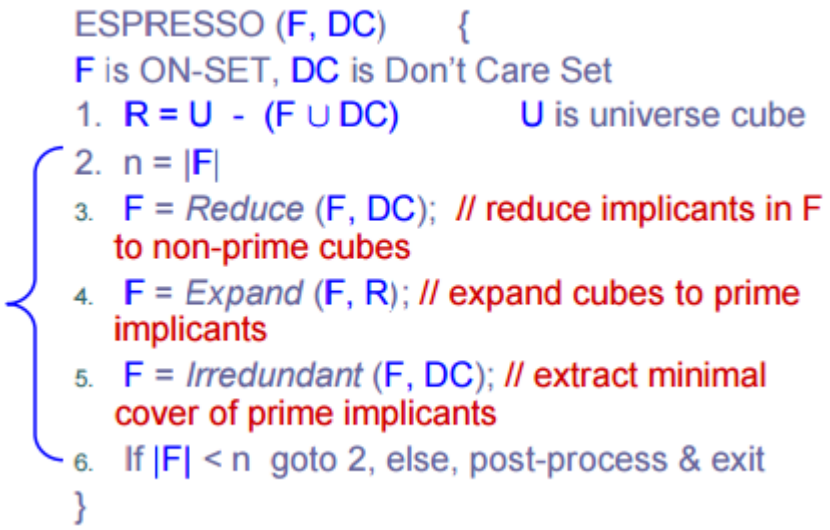

Рис.3.3. Алгоритм ESPRESSO [36]

3.2.4. Евристичний мінімізатор логіки ESPRESSO

Існує спеціальний мінімізатор логіки для апаратних засобів, розглянемо його і використовувані підходи у якості прикладу.

Всі цифрові системи складаються з двох елементарних функцій: елементів пам'яті для зберігання інформації і комбінаційних схем, що трансформують цю інформацію. Автомати, як і лічильники, є комбінацією елементів пам'яті і комбінаційних логічних схем. Оскільки елементи пам'яті є стандартними логічними схемами, вони вибираються з обмеженого набору альтернативних схем; таким чином, проектування

цифрових функцій зводиться до розробки комбінаційних схем затворів і їх взаємодії.

Відправною точкою для проектування цифрової логічної схеми є її бажана функціональність, отримана виходячи з аналізу системи в цілому, логічна схема якої є частиною. Опис може бути викладено в якійсь алгоритмічній формі або логічними рівняннями, але може бути узагальнений у вигляді таблиці. Наведений нижче приклад показує частину такої таблиці для 7-сегментного драйвера відображення, який переводить двійковий код для значень десяткової цифри в сигнали, які викликають освітлення відповідних сегментів дисплея.

Номер	Код	Сегменти	
		A B C D E F G	
0	0000	1 1 1 1 1 1 0	-A-
1	0001	0 1 1 0 0 0 0	| |
2	0010	1 1 0 1 1 0 1	F B
3	0011	1 1 1 1 0 0 1	| |
4	0100	0 1 1 0 0 1 1	-G-
5	0101	1 0 1 1 0 1 1	| |
6	0110	1 0 1 1 1 1 1	E C
7	0111	1 1 1 0 0 0 0	| |
8	1000	1 1 1 1 1 1 1	-D-
9	1001	1 1 1 1 0 1 1	

Процес реалізації починається з фази мінімізації логіки, щоб спростити таблицю функцій, об'єднавши окремі терми у більш великі, що містять менше змінних.

Далі, мінімізований результат може бути розділений на менші частини процедурою факторизації і в кінцевому підсумку відображається на доступні базові логічні комірки цільової технології. Ця операція зазвичай називається логічною оптимізацією [37].

3.2.5. BOOM - евристичний булевий мінімізатор

У 2003 було створено алгоритм дворівневої булевої мінімізації (BOOM) на основі нової парадигми генерації. На відміну від усіх попередніх методів мінімізації, де імпліканти генеруються знизу вгору, запропонований метод використовує підхід зверху вниз. Таким чином, замість збільшення розмірності імплікантів, опускаючи літерали з їхніх термів, розмірність терміна поступово зменшується шляхом додавання нових літералів.

Метод є вигідним, особливо для функцій з великою кількістю вхідних змінних (до тисяч) і коли визначено лише кілька термів, де інші інструменти мінімізації не застосовуються через тривалий час виконання. Метод був протестований на декількох різних видах проблем, і результати порівнювалися з ESPRESSO.

Зростання часу виконання для ESPRESSO і для BOOM показано на Рис.3.4., де кількість вхідних змінних вказано в дужках. Як бачимо, що хоча час виконання ESPRESSO зростає до 5000 с для 80 вхідних змінних, час виконання BOOM залишається майже постійним у межах використовуваної шкали для всіх розмірів. Вплив на час виконання ще більш наочно відображено на Рис.3.5., що показує відносне уповільнення BOOM та ESPRESSO, викликане Don’t-Care термами (DC) [38]. Ми бачимо, що відносне уповільнення BOOM для найвищого відсотка DC близько 7.5, тоді як для ESPRESSO - до 100.

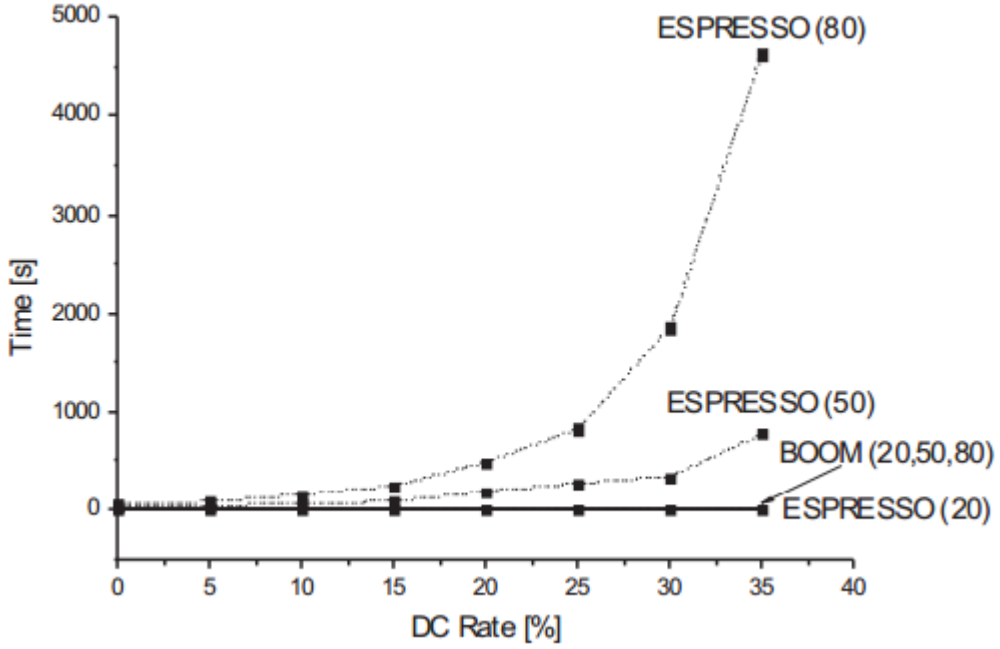

Рис.3.4. Витрати часу для ESPRESSO (пунктирні лінії) і BOOM (суцільна лінія) для різних відсотків DC [39]

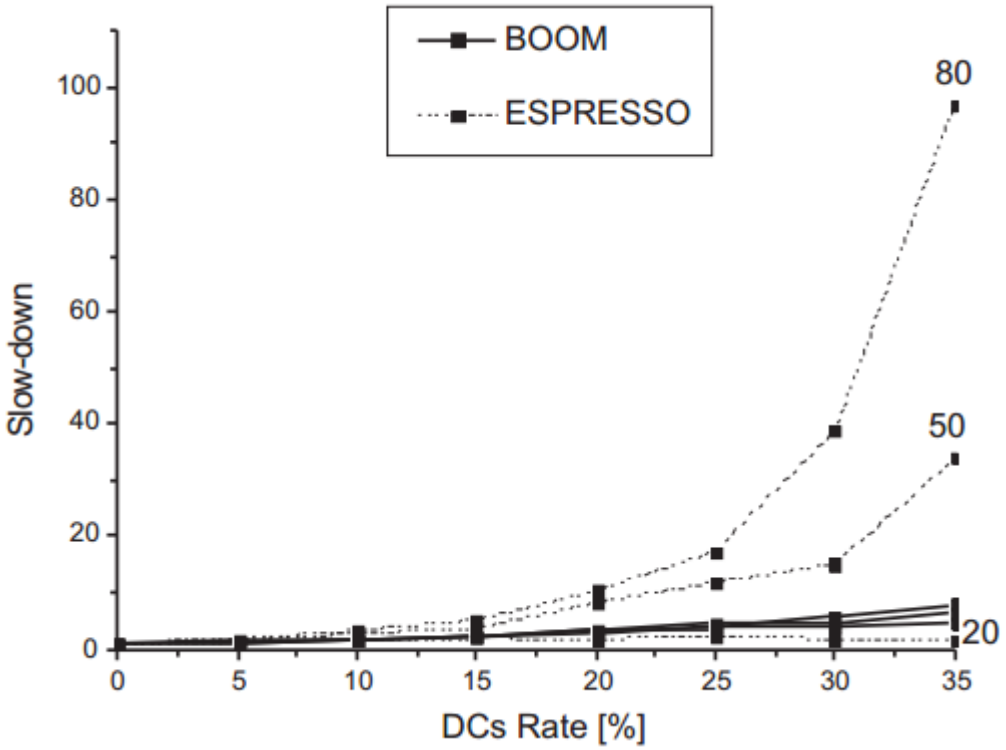

Рис.3.5. Відносне уповільнення [%] для різних відсотків DC [39]

З цього спостереження можна зробити висновок, що ESPRESSO надзвичайно чутливий до розмірності вихідних термів; час мінімізації

швидко зростає, при цьому зростаюча кількість вхідних даних не хвилює. З іншого боку, BOOM майже нечутливий до виміру термінів. Таким чином, BOOM може бути ефективно використаний для мінімізації функцій з великою часткою DC у вихідних термах. [39]

3.2.6. SAT-алгоритми для мінімізації логіки

На відміну від попередніх підходів, новий метод використовує розв'язувач SAT у якості базового рушія. Хоча загальна стратегія мінімізації методу базується на операторах, визначених у ESPRESSO-II, реалізація на основі SAT значно відрізняється. Мінімізатор SAT-ESPRESSO виявився в 5-20 разів швидше, ніж ESPRESSO-II, і в 3–5 разів швидше, ніж BOOM, на безлічі великих прикладів - див. Рис.3.6. [40]

Name	ESPRESSO-II	BOOM	SAT-ESPRESSO
50/100	17.79	8.48	3.04
50/150	48.40	31.57	6.69
50/200	138.55	109.03	23.13
100/50	9.23	0.63	3.11
100/200	1198.20	165.83	64.16
150/100	175.43	13.66	16.59
150/200	1320.30	1212.21	260.36
200/50	18.44	10.63	3.12
200/100	204.49	30.40	22.15
200/150	1265.68	186.52	56.05
200/200	2178.11	2626.39	134.19

Рис.3.6. Порівняння часу виконання (у секундах) [40]

SOP мінімізатор для логічних функцій

Швидкий і ефективний метод мінімізації для функцій, що описуються багатьма (до мільйонів) термами. Алгоритм заснований на обробці запропонованого ефективного подання набору термів - трійкового дерева. Досягнуто значне прискорення пошуку операції терму, відносно стандартного представлення табличних функцій. Процедура мінімізації

ґрунтується на швидкому застосуванні базових булевих операцій на трійковому дереві. Також підтримується мінімізація неповністю визначених функцій. Метод мінімізації був протестований на випадково створених великих сумах продуктів, згорнутих контрольних схем ISCAS і задачах SAT. Продуктивність запропонованого алгоритму порівнювалася з ESPRESSO. Знайдено дуже вигідне застосування алгоритму мінімізації - якщо він використовується для попередньої обробки функції, що має велику кількість термінів продукту, що виконується до ESPRESSO, то загальний час мінімізації значно зменшується, без зменшення якості результату [41].

n/c	TT-Min [s]	Espresso [s]	TT-Min+Espresso [s]	Improvement
50/218 (1)	0.08	50.40	40.84	19.0%
50/218 (2)	0.09	30.19	29.28	3.2%
50/218 (3)	0.09	36.13	34.54	4.4%
50/218 (10)	0.08	77.84	65.48	15.9%
75/325 (1)	0.30	22601.6	22029.2	2.5%
75/325 (10)	0.30	21554.1	18715.0	13%

Рис.3.7. Результати SAT [41]

3.2.7. Граф залежностей програми і його використання в оптимізації

Інша сфера де можна натрапити на потенційно корисні методи оптимізації, є оптимізації компіляторів. Серед них варто зазначити наступне проміжне програмне представлення, яке було названо графом залежностей програми (PDG), що робить явною як дані, так і контрольні залежності для кожної операції в програмі.

Залежності даних використовуються для представлення лише релевантних відносин потоку даних програми. Контрольні залежності аналогічно вводяться для представлення лише суттєвих взаємовідносин потоку керування програми. Контрольні залежності виводяться з звичайного графа потоку управління, див. Рис.3.8.

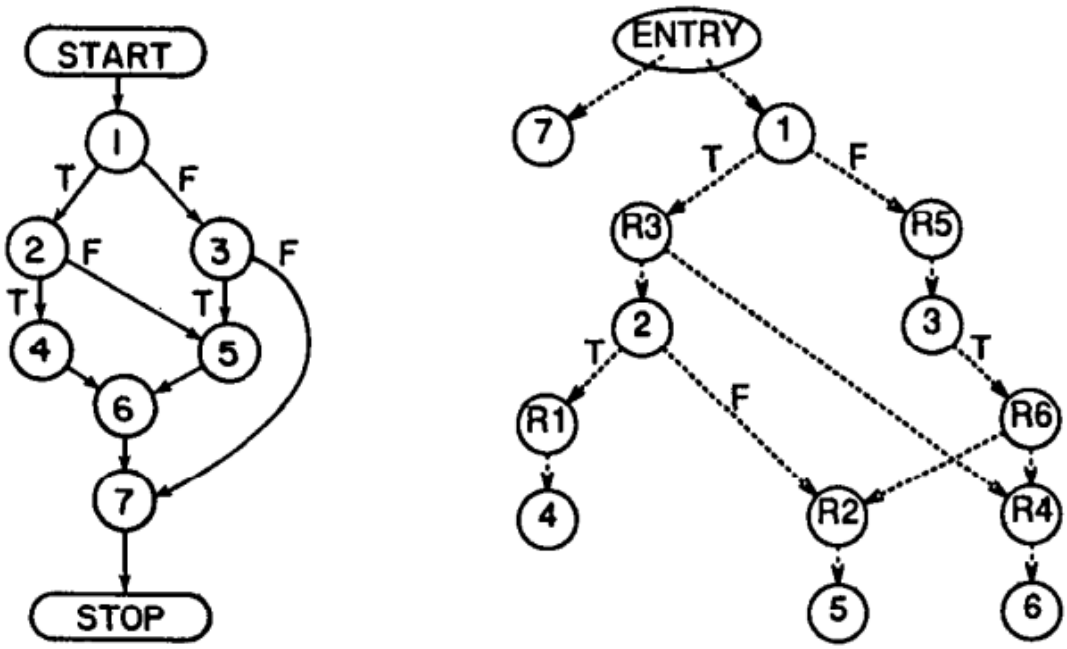

Рис.3.8. Графік управління потоком і його підграф контролю залежностей [42]

Багато традиційних оптимізацій працюють більш ефективно на PDG. Оскільки залежності в PDG з'єднують обчислювально пов'язані частини програми, одного проходу по цим залежностям достатньо для виконання більшості оптимізацій. PDG дозволяє трансформації, такі як векторизація, які раніше вимагали спеціальної обробки контрольної залежності, і вони виконуються у спосіб, що є єдиним для залежностей контролю і даних. Перетворення в програмі, які вимагають взаємодії двох типів залежностей, також легко обробляються PDG. Також він підтримує інкрементальну оптимізацію, що дозволяє запускати перетворення один за одним ii застосовувати оптимізацію тільки до залежностей. [42]

3.2.8. Порівняння алгоритмів

Таблиця 3.1 Порівняння можливостей алгоритмів

Можливість	Назва алгоритму							
	ABC-аналіз	карти Карно	метод Куайна — Мак-Класкі	ESPRESSO-II	BOOM	SAT-алгоритми	SOP	PDG
Швидкий пошук більшої частини покриття	+	+	-	-	-	-	-	-
Мінімізація покриття	-	+	+	+	+	+	+	-
Мінімізація не повністю визначених функцій	-	+	+	нема дани х	нема дани х	нема дани х	+	-
Визначення значимості елементу	+	-	-	-	-	-	-	-
Відносно просто запрограмувати	+	-	+	+	+-	+-	+-	-
Оптимальне використання ресурсів	нема дани х	не засто совн о	-	+-	+	+	+	не засто совн о
Найбільш ефективний на майштабах	не засто совн о	до 3-х термі в	до 4-х термі в	до 100 термі в	100-10.00 0 термі в	100-100.0 00 термі в	<10.0 00.00 0 термі в	не засто совн о

Оптимізація для інших алгоритмів	+	-	-	-	-	-	+-	+

Як видно з Табл.3.1, можна виділити 2 основних групи алгоритмів:

- Алгоритми попередньої оптимізації через пошук більшої частини покриття
- Алгоритми мінімізації покриття

У алгоритмів попередньої оптимізації є перевірений часом ABC-аналіз і його відносно просто запрограмувати, на відміну від його конкурента, тож будемо його використовувати для пришвидшення обчислень.

З мінімізацією покриття складніше, у зв'язку з наявністю, на перший погляд, переможця у вигляді ESPRESSO-II, однак він погано працює на великих об'ємах. Оптимально буде використати ESPRESSO-II на об'ємах до 100 термів, SAT-алгоритми на об'ємах від 100 до 10.000 термів і SOP для всього що буде більшим за 10.000 термів.

Виходячи з вищесказаного і доцільності, найкращими варіантами для реалізації виглядають зв'язки ABC+ESPRESSO-II, ABC+SAT і ABC+ESPRESSO-II+SAT. Їх і будемо використовувати.

ВИСНОВКИ ДО РОЗДІЛУ 3

Поставлено завдання оптимізації конфігурації яке може вирішуватися відомими математичними методами, такими як “метод гілок і меж” і “динамічного програмування”, що є частинами дискретної оптимізації. Задачами зі схожою проблематикою у дискретній оптимізації є задача пакування рюкзака [43] і задача про призначення [44].

Було розглянуто існуючі алгоритми з оптимізації і мінімізації, також, акцентовано увагу на деякі способи оптимізації що, на жаль, не застосовні в рамках хвилюючої нас проблеми, однак являють з себе гарний приклад того куди ще можна рухатись в алгоритмічному плані, для поліпшення оптимізаційних характеристик і, як наслідок, пришвидшення роботи оптимізатора.

Алгоритми, за сферою застосування, були розділені на попередньо оптимізаційні, і алгоритми мінімізації покриття. Підходящим в першій категорії виявився ABC-аналіз, чиє використання в оптимізаторі має зменшити загальний час витрачений на остаточну мінімізацію покриття. З алгоритмів мінімізації покриття було вирішено використовувати ESPRESSO-II і SAT для різної кількості термів. Перший вже має готові і перевірені часом реалізації, і на малих масштабах мало чим відрізняється від SAT, що дозволяє не зекономити ресурси на імплементацію і підтримку реалізації для малих систем.

РОЗДІЛ 4

РОЗРОБКА ПРОТОТИПУ СИСТЕМИ

Визначивши всі вимоги до системи і вирішивши питання як саме функціонуватиме кожна з її складових, можна приступити до розробки прототипу оптимізатора розподілених інформаційних систем.

4.1. Опис технологій

Звісно ж, виходячи з того що працювати доведеться з PIC, було б логічно використовувати технології що легко запускатимуться на будь-якій сучасній системі не заважаючи при цьому вже встановленим на них застосункам.

4.1.1. Docker

Docker - це комп'ютерна програма, яка виконує віртуалізацію на рівні операційної системи. [45] Вперше він був випущений в 2013 році і розроблений компанією Docker, Inc. [46].

Docker використовується для запуску пакетів програм, які називаються контейнерами. Контейнери відокремлені один від одного і з'єднують власні програми [47] інструменти, бібліотеки та конфігураційні файли; вони можуть спілкуватися один з одним через чітко визначені канали. Всі контейнери виконуються одним ядром операційної системи і, таким чином, є більш легкими, ніж віртуальні машини. Контейнери створюються зі зліпків (image), які вказують їх точний вміст. Зліпки часто створюються шляхом комбінування та модифікації стандартних зліпків, завантажених з відкритих сховищ. Більше про Docker можна дізнатись з мого короткого 15 хвилинного пояснення що було викладено у Youtube [48]

В даному випадку Docker цікавий тим, що ми можемо швидко відмасштабувати систему на безліч інстансів, обрахувати необхідне і тут

же схлопнути систему до мінімально-необхідної кількості екземплярів. До того ж, Docker дає можливість бути платформо-незалежними, єдина вимога - підтримка Linux namespaces і cgroups або їх замінників у вигляді тонких віртуальних машин на Windows чи MacOS.

4.1.2. Terraform

Terraform - інструмент з відкритим вихідним кодом що застосовується для опису інфраструктури кодом, створений HashiCorp. Вона дозволяє користувачам визначати та створювати інфраструктуру в центрі обробки даних за допомогою мови конфігурації високого рівня, відомої як мова налаштування Hashicorp (HCL), або, при необхідності, JSON. Terraform підтримує ряд постачальників хмарної інфраструктури, таких як веб-служби Amazon, IBM Cloud (раніше Bluemix), Google Cloud Platform, Linode, Microsoft Azure, інфраструктура Oracle Cloud або VMware vSphere, а також OpenStack.

HashiCorp також підтримує реєстр модулів Terraform, і кожен бажаючий може написати свій неофіційний провайдер до непідтримуваного Hashicorp продукту і користуватись ним для опису кодом абсолютно всього що має API.

В нашому випадку Terraform можна використовувати як проміжний інструмент, в описову мову якого можна витягнути існуючі елементи інфраструктури, а також, при наявності відповідних юзкейсів, можна пропонувати користувачу готовий фрагмент коду що описуватиме оптимальну інфраструктуру не тільки у вигляді таблиці ресурсів і ресурсних груп, а й у вигляді коду.

4.1.3. Terragrunt

Terragrunt - це тонка обгортка для Terraform, яка надає додаткові інструменти для перевикористання вже написаного Terraform коду,

роботи з декількома модулями Terraform і керуванням станом інфраструктури ло лежить не локально.

Використовуватиметься тих випадках, де юзкейси досить складні і тому покриті не тільки Terraform кодом, а й ще й загорнуті в Terragrunt, або вихідна інфраструктура буде описана з його допомогою - потрібно буде виконати terragrunt plan для діставання поточної конфігурації кожного з елементів яка може бути розмазана на декількох рівнях що один від одного наслідується.

4.1.4. Python 3

Python є інтерпретованою мовою програмування високого рівня загального призначення. Створений Гвідо ван Россумом і вперше випущений в 1991 році, філософія дизайну Python підкреслює читабельність коду з його помітним використанням значних пробілів. Його мовні конструкції та об'єктно-орієнтований підхід спрямовані на те, щоб допомогти програмістам писати чіткий, логічний код для малих і великих проектів.

У Python наявні динамічна типізація і збірка сміття. Він підтримує багато парадигм програмування, включаючи процедурні, об'єктно-орієнтовані та функціональні стилі.

Власне, враховуючи що ми використовуємо Docker, мова на якій буде реалізовуватись функціонал оптимізатора є відносно не важливою, однак вибір був зроблений на Python у зв’язку з величезною екосистемою бібліотек, в тому числі і з алгоритмами і машиного навчання, частина з яких буде застосована у прототипі, а інша може стати в нагоді при розробці повноціної системи.

Використовуватиметься тільки діалект Python 3, у зв’язку з тим що Python 2 перестане отримувати оновлення вже у 2020 році [49]

ВИСНОВКИ ДО РОЗДІЛУ 4

Сформульовано вимоги до технологічного стеку за допомогою якого можна досягти поставленої мети в найкоротші терміни і задовольнити описані технічні і функціональні можливості.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Harari Y. Sapiens: A Brief History of Humankind. / Y. Harari // Harper. ISBN 0062316095 — 2015. — Chapter 10: The Scent of Money

2. Методи захисту інформаціїї в комп'ютерних мережах [Електронний ресурс] / М.Г.Кузнєцова — 2006. — Режим доступу до ресурсу: http://dspace.nbuv.gov.ua/bitstream/handle/123456789/50851/06-Kuznetsova.pdf?sequence=1

3. Comparing Data Center Costs With Public IaaS Cloud Services [Електронний ресурс] / Nik Simpson — 2012. — Режим доступу до ресурсу: https://www.gartner.com/doc/2048419/comparing-data-center-costs-public

4. The Financial Case for Moving to the Cloud [Електронний ресурс] / Sanil Solanki, Michael Smith, Tomas Nielsen — 2015. — Режим доступу до ресурсу: https://www.gartner.com/doc/3030826/financial-case-moving-cloud

5. Patterns of Enterprise Application Architecture [Електронний ресурс] / Martin Fowler — 2003. — Режим доступу до ресурсу: http://www.pearsonhighered.com/educator/academic/product/0,3110,0321127420,00.html

6. Software Architecture in Practice, 3nd Edition [Електронний ресурс] / L. Bass, P. Clements, R. Kazman — 2013. — Режим доступу до ресурсу: https://www.pearson.com/us/higher-education/program/Bass-Software-Architecture-in-Practice-3rd-Edition/PGM317124.html

7. SOFSEM 2010: Theory and Practice of Computer Science, 36th [Електронний ресурс] / J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorny, B. Rumpe — 2010. — Режим доступу до ресурсу: http://www.springer.com/us/book/9783642112652#

8. Fault Tolerance Mechanisms in Distributed Systems [Електронний ресурс] / A. Sari, M. Akkaya — 2015. — Режим доступу до ресурсу: https://www.researchgate.net/publication/287198069_Fault_Tolerance_Mechanisms_in_Distributed_Systems

9. Hookway B. Interface. / Hookway B. // Chapter 1: The Subject of the Interface. MIT Press. ISBN 9780262525503 — 2014. — С. 1—58.

10. Web Services Architecture: Relationship to the World Wide Web and REST Architectures [Електронний ресурс] / D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard — 2004. — Режим доступу до ресурсу: https://www.w3.org/TR/ws-arch/

11. Standards, compliance, and Rational Unified Process [Електронний ресурс] / W. Cottrell — 2004. — Режим доступу до ресурсу: https://www.ibm.com/developerworks/rational/library/4763.html

12. Iterative development illustration [Електронний ресурс] / Dutchguilder — 2007. — Режим доступу до ресурсу: https://commons.wikimedia.org/wiki/File:Development-iterative.png

13. Hotline.ua [Електронний ресурс] — Режим доступу до ресурсу: https://hotline.ua/mobile/mobilnye-telefony-i-smartfony/

14. AWS to Azure services comparison [Електронний ресурс] / M. Wasson, A. Buck, C. Bennage та ін. — 2019. — Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services

15. AWS to Azure services comparison source code [Електронний ресурс] / M. Wasson, A. Buck, C. Bennage та ін. — 2019. — Режим доступу до ресурсу: https://github.com/microsoftdocs/architecture-center/blob/master/docs/aws-professional/services.md

16. Announcing Azure to AWS migration support in AWS Server Migration Service [Електронний ресурс] — 2019. — Режим доступу до ресурсу: https://aws.amazon.com/about-aws/whats-new/2019/04/announcing_azure_awsmigration_servermigrationservice/

17. CompareCloud [Електронний ресурс] / Ilyas F. — 2019. — Режим доступу до ресурсу: http://comparecloud.in/

18. CompareCloud source code [Електронний ресурс] / Ilyas F. — 2019. — Режим доступу до ресурсу: https://github.com/ilyas-it83/CloudComparer/

19. Cloud Comparison Tool [Електронний ресурс] — Режим доступу до ресурсу: https://www.cloudcomparisontool.com/

20. Migrating 23TB from S3 to B2 in just 7 hours [Електронний ресурс] / G. R. Sudderth — 2019. — Режим доступу до ресурсу: https://nodecraft.com/blog/development/migrating-23tb-from-s3-to-b2-in-just-7-hours

21. AWS Solutions [Електронний ресурс] — 2019. — Режим доступу до ресурсу: https://aws.amazon.com/solutions/

22. Azure solution architectures [Електронний ресурс] — 2019. — Режим доступу до ресурсу: https://azure.microsoft.com/en-us/solutions/architecture/

23. Google Cloud Solutions Architecture Reference [Електронний ресурс] — 2019. — Режим доступу до ресурсу: http://gcp.solutions/

24. Optimize cloud architecture and spend to continuously improve your modern cloud environment [Електронний ресурс] — Режим доступу до ресурсу: https://docs.newrelic.com/docs/using-new-relic/welcome-new-relic/optimize-your-cloud-native-environment/optimize-cloud-architecture-spend-continuously-improve-your-modern-cloud-environment

25. CloudCheckr + Datadog: Better rightsizing of cloud resources [Електронний ресурс] / A. L. McCloud — 2017. — Режим доступу до ресурсу: https://www.datadoghq.com/blog/rightsizing-cloudcheckr/

26. Entrepreneurship and the U.S. Economy [Електронний ресурс] — 2016. — Режим доступу до ресурсу: https://www.bls.gov/bdm/entrepreneurship/entrepreneurship.htm

27. How likely is your startup to fail within a year? Here's the bitter pill [Електронний ресурс] / M. Velayanikal — 2016. — Режим доступу до ресурсу: https://www.techinasia.com/startup-failure-analysis

28. Multiclass classification [Електронний ресурс] — 2019. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Multiclass_classification

29. Multi-label classification [Електронний ресурс] — 2018. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Multi-label_classification

30. Cloud Comparer data [Електронний ресурс] / Ilyas F., Vidyasagar Machupalli — 2017. — Режим доступу до ресурсу: https://github.com/ilyas-it83/CloudComparer/raw/master/data/CloudComparer.xlsx

31. Working with Excel Files in Python [Електронний ресурс] / C. Withers, C. Clark — 2016. — Режим доступу до ресурсу: http://www.python-excel.org/

32. ABC class [Електронний ресурс] / Masaqui — 2011. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/File:ABC_class.jpg

33. ABC-аналіз [Електронний ресурс] — 2013. — Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/ABC-%D0%B0%D0%BD%D0%B0%D0%BB%D1%96%D0%B7

34. Douglas L. Design of Logic Systems / L. Douglas // Van Nostrand (UK). ISBN 0-442-30606-7 — 1985.

35. Nelson V.P. Digital Circuit Analysis and Design / V.P. Nelson // Prentice Hall. ISBN 978-0134638942 — 1995. — С. 234.

36. Fundamental Algorithms for System Modeling, Analysis, and Optimization [Електронний ресурс] / E. A. Lee, J. Roychowdhury, S. A. Seshia — 2011. — Режим доступу до ресурсу: https://ptolemy.berkeley.edu/projects/embedded/eecsx44/lectures/Fall2013/BooleanBasicsLogicOptimization-1.pdf

37. De Micheli, G. Synthesis and Optimization of Digital Circuits / G. De Micheli // McGraw-Hill Science Engineering. ISBN 0-07-016333-2 — 1994.

38. Strong J.A. Basic Digital Electronics / J.A. Strong // Springer — 2013 ISBN 940113118X — С. 28—29.

39. Fisher P. BOOM — A HEURISTIC BOOLEAN MINIMIZER / P. Fisher, J. Hlavicka [Електронний ресурс] — 2002. — Режим доступу до ресурсу: http://www.cai.sk/ojs/index.php/cai/article/viewFile/450/356

40. Sapra S. SAT-Based Algorithms for Logic Minimization [Електронний ресурс] / S. Sapra, M. Theobald, E. Clarke — Режим доступу до ресурсу: https://pdfs.semanticscholar.org/67fd/d3e15a500ebd2b10c6175ee344274d4e1d6a.pdf

41. Fisher P. A Fast SOP Minimizer for Logic Functions Described by Many Product Terms [Електронний ресурс] / P. Fisher, D. Toman — Режим доступу до ресурсу: https://ddd.fit.cvut.cz/prj/TT-Min/dsd09_Toman.pdf

42. Ferrante J. The Program Dependence Graph and Its Use in Optimization [Електронний ресурс] / J. Ferrante, K. J. Ottenstein, J. D. Warren // ACM Transactions on Programming Languages and Systems — 1987. — Vol. 9, №3 — Режим доступу до ресурсу: https://www.cs.utexas.edu/~pingali/CS395T/2009fa/papers/ferrante87.pdf

43. Kellerer H. Knapsack Problems / H. Kellerer, U. Pferschy, D. Pisinger // Springer-Verlag Berlin Heidelberg ISBN 978-3-642-07311-3 — doi:10.1007/978-3-540-24777-7 — 2004. — С. 548.

44. Richard A., B. Combinatorial matrix classes. / B. A. Richard // Encyclopedia of Mathematics and Its Applications. Cambridge: Cambridge University Press. ISBN 0-521-86565-4. Zbl 1106.05001. — 2006. — С. 108.

45. Maureen O. Ben Golub, Who Sold Gluster to Red Hat, Now Running dotCloud / O. Maureen // SYS-CON Media. — 2013.

46. Ratan V. Docker: A Favourite in the DevOps World / V. Ratan — 2017.

47. Linville M. Docker frequently asked questions (FAQ) [Електронний ресурс] / M. Linville, R. Anderson — 2018. — Режим доступу до ресурсу: https://docs.docker.com/engine/faq/

48. Vlasov M. //brief Docker #1 [Електронний ресурс] / M. Vlasov, K. Stativkin — 2018. — Режим доступу до ресурсу: https://youtu.be/TilqjYw-WzA

49. Peterson B. PEP 373 -- Python 2.7 Release Schedule [Електронний ресурс] / B. Peterson — 2008. — Режим доступу до ресурсу: https://www.python.org/dev/peps/pep-0373/

File: .generate.yaml

```
version: '3'

services:
  ## Your app. Please, don't rename.
  #app:
  #  build: .
  #  restart: always
  #  environment:
  #    SERVICE_REGION: $SERVICE_REGION
  #    SERVICE_NAME: $SERVICE_NAME
  #    SERVICE_ENV: $SERVICE_ENV
  #    CONSUL_HTTP_ADDR: $CONSUL_HTTP_ADDR
  #  depends_on:
  #    - add_keys_to_consul
  #  volumes:
  #    - .:/app
  #    - ./docker/logs/:$SERVICE_LOG_PATH
  #  labels:
  #    - "traefik.backend=app"
  #    - "traefik.frontend.rule=Host:app.localhost"


  # Add Key/Value to Consul.
  add_keys_to_consul: !include:add_keys_to_consul.yaml |

  # Key/Value storage with config for our app and other infrastructure.
  consul: !include:consul.yaml |

  # Search engine and document-based DB.
  elasticsearch: !include:elasticsearch.yaml |

  # Add logs from filesystem to Elasticsearch.
  filebeat: !include:filebeat.yaml |

  # Visualization tool for data in Elasticsearch.
  kibana: !include:kibana.yaml |

  # Queue with logs whish should write to DB.
  rabbit: !include:rabbit.yaml |

  # Redis database
  redis: !include:redis.yaml | # by default: 3.2.4-alpine

  # Consul services registrator.
  registrator: !include:registrator.yaml |

  # MySQL & MariaDB section.
  # If change image - please reinit service (docker-compose rm)
```

```
  mysql: !include:mysql.yaml | # by default: 5.7.17, DB: appdb, L: root, P:
mysql_root
    #image: mysql:5.6.35
    #image: mysql:5.5.53

  mariadb: !include:mariadb.yaml | # by default: 10.1.23, DB: appdb, L: root,
P: mysql_root
    #image: mariadb:10.0.31

  # Traefik make access to services by `SERVICE.localhost`
  traefik: !include:traefik.yaml |


volumes:
  rabbitdb:
    external: true
  consuldb:
    external: true

networks:
  default:
    external:
      name: dev-compose-network
```

File: .docker_compose_generator/consul/app-logs.json

```
{
    "SERVICE_REGION": {
        "apps": {
            "SERVICE_NAME": {
                "SERVICE_ENV": {
                    "logs_TTL": 48
                }
            }
        },
        "filebeat": {
            "SERVICE_CLUSTER": {
                "SERVICE_NAME": {
                    "doc_type": "SERVICE_NAME",
                    "json": 1,
                    "path": "/var/log/SERVICE_NAME/*.json"
                }
            }
        }
    }
}
```

File: .docker_compose_generator/consul/infrastructure.json

```
{
```

```
    "us-east-1": {
        "local": {
            "dev-elastic-log": {
                "host": "elasticsearch",
                "port": 9200
            },
            "rabbitmq": {
                "host": "rabbit",
                "port": 5672
            },
            "influxdb": {
                "host": "influx",
                "port": 8086,
                "protocol": "http"
            },
            "mail-api": {
                "api-test-host": null,
                "api-test-scheme": null
            }
        },
        "AWS-services": {
            "elasticache": {
                "redis": {
                    "redisdb": {
                        "host": "redis",
                        "port": 6379
                    }
                },
                "memcached": {
                    "memcachedb": {
                        "host": "memcached",
                        "port": 11211
                    }
                }
            },
            "RDS": {
                "mysql" : {
                    "mysqldb": {
                        "host": "mysql",
                        "port": 3306
                    }
                }
            }
        }
    }
}
```

File:
.docker_compose_generator/yaml_files/add_keys_to_consul/add_keys_to_consul.py

```
import json
from urllib.error import HTTPError
from os import environ as env
# consul_kv for work with key-value storage in Consul
from consul_kv import Connection

with open('/app/consul/consul.json') as json_data:
    APP = {env['SERVICE_REGION']: \
            {'apps': \
                {env['SERVICE_NAME']: \
                    {env['SERVICE_ENV']: \
                        json.load(json_data) \
           }}}}

try:
    with open('/app/consul/consul-private.json') as json_data:
        APP_PRIVATE = {env['SERVICE_REGION']: \
            {'apps': \
                {env['SERVICE_NAME']: \
                    {env['SERVICE_ENV']: \
                        json.load(json_data) \
            }}}}
except FileNotFoundError:
    APP_PRIVATE = {}

with open('/app/consul/app-logs.json') as json_data:
    APP_LOGS = json.load(json_data)

with open('/app/consul/infrastructure.json') as json_data:
    INFRASTRUCTURE = json.load(json_data)


CONN = Connection(endpoint='http://' + env['CONSUL_HTTP_ADDR'] + '/v1/kv')

try:
    CONN.delete(env['SERVICE_REGION'] + '/apps/' + env['SERVICE_NAME'] + '/'
+ env['SERVICE_ENV'], recurse=True)
except HTTPError:
    pass

CONN.put_dict(APP)
CONN.put_dict(APP_PRIVATE)
CONN.put_dict(APP_LOGS)
CONN.put_dict(INFRASTRUCTURE)
print('Remove old and add new ' + env['SERVICE_NAME'] + ' configs.')
```

File: .docker_compose_generator/yaml_files/add_keys_to_consul/Dockerfile

```
FROM frolvlad/alpine-python3
```

```
MAINTAINER MaxymVlasov

COPY . /app

RUN pip3 install
https://pypi.python.org/packages/e0/98/5003c51b50aedbad6dee49b49d359ef0f03ef4
947b2ae81f59c43d84c149/consul_kv-0.4.tar.gz \
 && chmod +x /app/wait_infrastructure.sh

ENV SERVICE_CLUSTER=cluster_name \
    SERVICE_ENV=dev \
    SERVICE_NAME=app \
    SERVICE_REGION=us-east-1

WORKDIR /app
CMD ["/app/wait_infrastructure.sh"]
```

File:
.docker_compose_generator/yaml_files/add_keys_to_consul/wait_infrastructure.s
h

```
#!/bin/sh
# wait_infrastructure.sh

until wget ${CONSUL_HTTP_ADDR}; do
  >&2 echo "Consul is unavailable - sleeping"
  sleep 1
done

>&2 echo "Consul is up - add keys to Consul"

# Remove temporary files
rm index.html

# Move consul configs from mounted docker volumes to dir into container,
# because original files should not changed.
rm -rf /app/consul/
cp -r /consul/ /app/consul/
cp /consul-dev/consul.json /app/consul/consul.json
cp /consul-dev/consul-private.json /app/consul/consul-private.json
2>/dev/null

/app/go-replace -s 'SERVICE_CLUSTER' -r ${SERVICE_CLUSTER} /app/consul/app-
logs.json
/app/go-replace -s 'SERVICE_ENV' -r ${SERVICE_ENV} /app/consul/app-logs.json
/app/go-replace -s 'SERVICE_NAME' -r ${SERVICE_NAME} /app/consul/app-
logs.json
/app/go-replace -s 'SERVICE_REGION' -r ${SERVICE_REGION} /app/consul/app-
logs.json
```

```
python3 /app/add_keys_to_consul.py
```

File: .docker_compose_generator/yaml_files/filebeat/Dockerfile

```
FROM maxymvlasov/filebeat-consul:latest

MAINTAINER MaxymVlasov

COPY /go-replace /usr/local/bin/goreplace
COPY /template.ctmpl /provision.sh /etc/filebeat/

RUN chmod +x /etc/filebeat/provision.sh

ENV ELASTIC="'elasticsearch:9200'" \
    CLUSTER_NAME=cluster_name \
    SERVICE_REGION=us-east-1 \
    SERVICE_NAME=app \
    SERVICE_ENV=dev \
    SERVICE_LOG_PATH=/var/log/app/
```

File: .docker_compose_generator/yaml_files/filebeat/provision.sh

```
#!/bin/sh

/usr/local/bin/goreplace -s 'ELASTIC' -r ${ELASTIC}
/etc/filebeat/template.ctmpl
/usr/local/bin/goreplace -s 'CLUSTER_NAME' -r ${CLUSTER_NAME}
/etc/filebeat/template.ctmpl
/usr/local/bin/goreplace -s 'SERVICE_REGION' -r ${SERVICE_REGION}
/etc/filebeat/template.ctmpl
/usr/local/bin/goreplace -s 'SERVICE_LOG_PATH' -r ${SERVICE_LOG_PATH}
/etc/filebeat/template.ctmpl

/usr/local/bin/goreplace -s 'SERVICE_REGION' -r ${SERVICE_REGION}
/etc/filebeat/clear.ctmpl
/usr/local/bin/goreplace -s 'SERVICE_NAME' -r ${SERVICE_NAME}
/etc/filebeat/clear.ctmpl
/usr/local/bin/goreplace -s 'SERVICE_ENV' -r ${SERVICE_ENV}
/etc/filebeat/clear.ctmpl

/usr/local/bin/consul-template --consul-addr=${CONSUL_HTTP_ADDR} \
  --template "/etc/filebeat/template.ctmpl:/etc/filebeat/filebeat.yml" \
  --template
"/etc/filebeat/clear.ctmpl:/etc/crontabs/clear:/etc/filebeat/after_consul.sh"
```

File: .docker_compose_generator/yaml_files/filebeat/template.ctmpl

```
###################### Filebeat Configuration #########################
#============================ Filebeat prospectors
=============================
filebeat.prospectors:
- input_type: log
```

```
  paths:
  - SERVICE_LOG_PATH/*.json
  document_type: nginx-access
  json:
    -json.keys_under_root: true
    -json.add_error_key: true
    -json.message_key: message
#================================ General
=====================================

# The name of the shipper that publishes the network data. It can be used to
group
# all the transactions sent by a single shipper in the web interface.
name: ""

# The tags of the shipper are included in their own field with each
# transaction published.
tags: ["denise", "json"]

# Optional fields that you can specify to add additional information to the
# output.
fields:
  env: dev

#================================ Outputs
=====================================
#------------------------- Elasticsearch output ----------------------------
---
output.elasticsearch:
  enabled: true

  hosts: [ ELASTIC ]

  #compression_level: 6

  # Template name. By default the template name is filebeat.
  template.name: "filebeat"

  # Path to template file
  template.path: "/etc/filebeat/filebeat.template.json"

  # Overwrite existing template
  template.overwrite: true

  # If set to true, filebeat checks the Elasticsearch version at connect
time, and if it
  # is 2.x, it loads the file specified by the template.versions.2x.path
setting. The
  # default is true.
```

```
  template.versions.2x.enabled: false

  # Path to the Elasticsearch 2.x version of the template file.
  template.versions.2x.path: "/etc/filebeat/filebeat.template-es2x.json"

#================================ Logging
=====================================
logging.level: debug
logging.selectors: ["*"]
```

File: .docker_compose_generator/yaml_files/add_keys_to_consul.yaml

```
build:
DOCKER_COMPOSE_GENERATOR_PATH/.docker_compose_generator/yaml_files/add_keys_t
o_consul
restart: on-failure
environment:
  CONSUL_HTTP_ADDR: $CONSUL_HTTP_ADDR
  SERVICE_CLUSTER: $SERVICE_CLUSTER
  SERVICE_ENV: $SERVICE_ENV
  SERVICE_NAME: $SERVICE_NAME
  SERVICE_REGION: $SERVICE_REGION
volumes:
  - DOCKER_COMPOSE_GENERATOR_PATH/.docker_compose_generator/consul/:/consul/
  - ./docker/:/consul-dev/
depends_on:
  - consul
```

File: .docker_compose_generator/yaml_files/consul.yaml

```
image: consul:0.8.3
restart: always
ports:
  - "8400:8400"
  - "8500:8500"
  - "8600:53/udp"
expose:
  - 8300
  - 8301
  - 8302
command: agent -server -bootstrap-expect=1 -advertise=172.19.0.3 -ui -
client=0.0.0.0
labels:
  - "traefik.backend=consul"
  - "traefik.frontend.rule=Host:consul.localhost"
volumes:
  - consuldb:/consul/data
```

File: .docker_compose_generator/yaml_files/elasticsearch.yaml

```
image: docker.elastic.co/elasticsearch/elasticsearch:5.4.0
restart: always
```

```
environment:
  - cluster.name=ESCluster
  - bootstrap.memory_lock=true
  - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  - xpack.security.enabled=false
  - node.name=es_node1
ulimits:
  memlock:
    soft: -1
    hard: -1
expose:
  - 9200
labels:
  - "traefik.backend=elasticsearch"
  - "traefik.frontend.rule=Host:elasticsearch.localhost"
```

File: .docker_compose_generator/yaml_files/filebeat.yaml

```
build:
DOCKER_COMPOSE_GENERATOR_PATH/.docker_compose_generator/yaml_files/filebeat
restart: always
environment:
  CONSUL_HTTP_ADDR: $CONSUL_HTTP_ADDR
  SERVICE_NAME: $SERVICE_NAME
  SERVICE_LOG_PATH: $SERVICE_LOG_PATH
volumes:
  - ./docker/logs/:SERVICE_LOG_PATH
depends_on:
  - elasticsearch
```

File: .docker_compose_generator/yaml_files/kibana.yaml

```
image: docker.elastic.co/kibana/kibana:5.4.0
restart: always
environment:
  ELASTICSEARCH_URL: http://elasticsearch:9200
  SERVER_NAME: http://kibana.localhost
  XPACK_MONITORING_UI_CONTAINER_ELASTICSEARCH_ENABLED: 'false'
labels:
  - "traefik.backend=kibana"
  - "traefik.frontend.rule=Host:kibana.localhost"
```

File: .docker_compose_generator/yaml_files/mariadb.yaml

```
image: mariadb:10.1.23
restart: always
environment:
  MYSQL_ROOT_PASSWORD: mysql_root
  MYSQL_DATABASE: appdb
labels:
  - "traefik.enable=false"
```

File: .docker_compose_generator/yaml_files/mysql.yaml

```
image: mysql:5.7.17
restart: always
environment:
  MYSQL_ROOT_PASSWORD: mysql_root
  MYSQL_DATABASE: appdb
labels:
  - "traefik.enable=false"
```

File: .docker_compose_generator/yaml_files/rabbit.yaml

```
restart: always
image: maxymvlasov/rabbitmq:latest
hostname: rabbitmq
domainname: rabbit.local
expose:
  - 4369
  - 5671
  - 5672
  - 15671
  - 15672
  - 25672
volumes:
  - rabbitdb:/var/lib/rabbitmq
labels:
  - "traefik.backend=rabbit"
  - "traefik.backend.port=15672"
  - "traefik.frontend.rule=Host:rabbit.localhost"
```

File: .docker_compose_generator/yaml_files/redis.yaml

```
image: redis:3.2.4-alpine
restart: always
expose:
  - 6379
```

File: .docker_compose_generator/yaml_files/registrator.yaml

```
image: gliderlabs/registrator:latest
restart: always
volumes:
  - /var/run/docker.sock:/tmp/docker.sock
command: consul://consul:8500
depends_on:
  - consul
```

File: .docker_compose_generator/yaml_files/traefik.yaml

```
image: traefik
command: --web --docker --docker.domain=localhost --logLevel=DEBUG
ports:
  - 80:80
  - 8080:8080
```

```
volumes:
  - /var/run/docker.sock:/var/run/docker.sock
  - /dev/null:/traefik.toml
labels:
  - "traefik.backend=traefik"
  - "traefik.frontend.rule=Host:traefik.localhost"
```

File: .docker_compose_generator/Dockerfile

```
FROM frolvlad/alpine-python3

MAINTAINER MaxymVlasov

RUN pip3 install pyyaml

COPY / /loader
CMD ["python3", "/loader/loader.py"]
```

File: .docker_compose_generator/loader.py

```
import yaml
import os

YAML_FILES_DIR = '/loader/yaml_files'
CONSTRUCTOR_PREFIX = '!include:'

class SafeLoaderMeta(type):

    def __new__(metacls, __name__, __bases__, __dict__):
        """Add include constructer to class."""

        # register the include constructor on the class
        cls = super().__new__(metacls, __name__, __bases__, __dict__)

        for filename in os.listdir(YAML_FILES_DIR):
            if os.path.splitext(filename)[1].lstrip('.')  == 'yaml' or \
               os.path.splitext(filename)[1].lstrip('.')  == 'yml':
                cls.add_constructor(CONSTRUCTOR_PREFIX + filename,
cls.construct_include)

        return cls


class SafeLoader(yaml.SafeLoader, metaclass=SafeLoaderMeta):
    """YAML Loader with custom constructors."""

    def __init__(self, stream):
        """Initialise Loader."""

        self._root = YAML_FILES_DIR
```

```
        super().__init__(stream)

    def construct_include(self, node):
        """Include file referenced at node."""

        filename = os.path.abspath(os.path.join(
            self._root, node.tag.replace(CONSTRUCTOR_PREFIX, '')
        ))
        extension = os.path.splitext(filename)[1].lstrip('.')

        with open(filename, 'r') as file_name:
            if extension in ('yaml', 'yml'):

                predefined_data = yaml.safe_load(file_name)

                for key, value in predefined_data.items() :
                    if key == 'build':
                        predefined_data[key] =
value.replace('DOCKER_COMPOSE_GENERATOR_PATH', \

os.environ['DOCKER_COMPOSE_GENERATOR_PATH'])
                    elif key == 'volumes':
                        for index, item in enumerate(predefined_data[key]):
                            predefined_data[key][index] =
item.replace('DOCKER_COMPOSE_GENERATOR_PATH', \

os.environ['DOCKER_COMPOSE_GENERATOR_PATH']) \

.replace('SERVICE_LOG_PATH', \

os.environ['SERVICE_LOG_PATH'])


                overridden_data = yaml.safe_load(node.value)

                try:
                    data = {**predefined_data, **overridden_data}
                except TypeError: # Arises when overridden_data is empty
                    data = predefined_data

                return data

            else:
                return ''.join(data.readlines())


if __name__ == '__main__':
    try:
        with open('/app/.generate.yaml', 'r') as generate_yaml:
```

```
            data = yaml.safe_dump(yaml.load(generate_yaml, SafeLoader),
default_flow_style=False)
    except FileNotFoundError:
        START_COLOR = '\033[36m'
        END_COLOR = '\033[0m'

        print(START_COLOR)
        print('=====================================================')
        print('  Firstly, you need add .generate.yaml to your repo')
        print('=====================================================')
        print(END_COLOR)
        exit()

    with open('/app/docker-compose.yaml', 'w') as docker_compose:
        docker_compose.write(data)
```